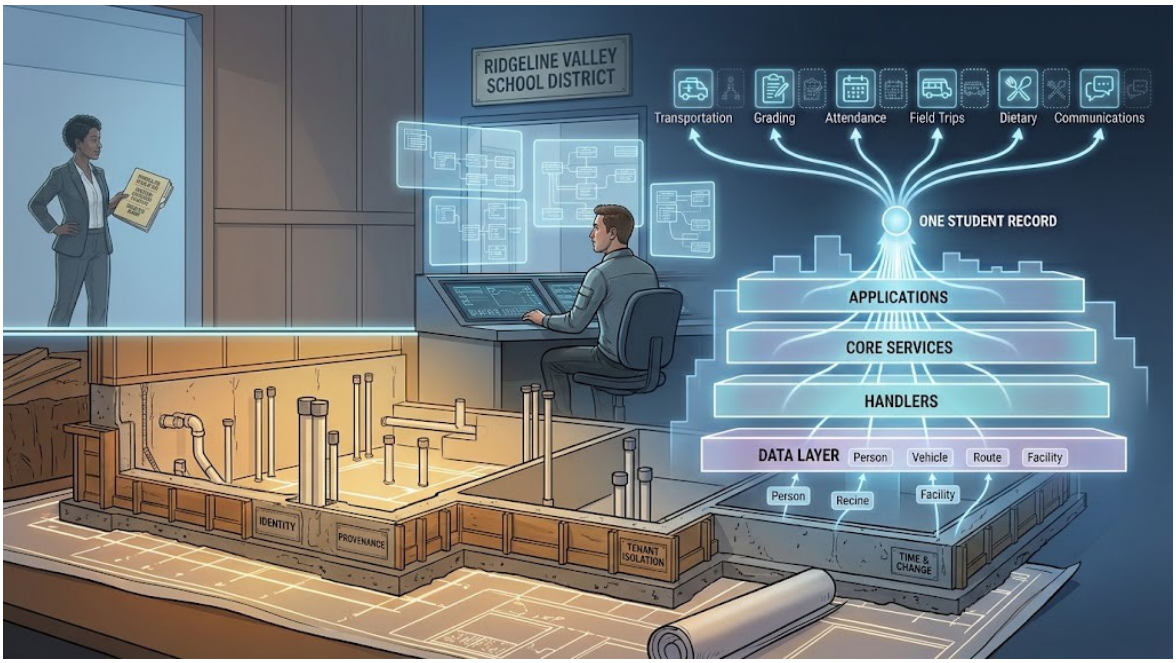


Constellation Programming

Why Now

Software has always been constrained by a basic human limitation: the more sophisticated the thing we want to build, the more kinds of specialized knowledge we have to assemble around it.

A sophisticated software system may require expertise in Python, JavaScript, Swift, SQL, cloud infrastructure, databases, security, APIs, networking, interface design, payment systems, testing, deployment, analytics, and dozens of other technologies and disciplines.

And surface-level knowledge is not enough.

If a system is expected to perform reliably at scale, each of those layers has to be implemented with sufficient depth and then coordinated with all the others. Historically, that has meant teams of specialists, organizational layers, specifications, meetings, handoffs, testing processes, and enormous amounts of coordination.

The complexity of the software therefore imposed a practical constraint on the size of the thing an individual could reasonably attempt to create.

Artificial intelligence begins to change that constraint.

For the first time, we have access to something approaching broadly available technical intelligence: a system capable of operating across programming languages, architectures,

frameworks, disciplines, and bodies of knowledge that no single human could realistically master at comparable breadth.

That does not mean an AI model can simply be handed a sentence and reliably produce any sophisticated software system we can imagine.

Intelligence alone is not enough.

A library containing every relevant book does not automatically produce a great novel.

The intelligence has to be given context. The work has to be decomposed. Decisions have to persist. One task has to feed the next. Outputs have to be inspected. Errors have to be corrected. Different technical capabilities have to remain coordinated around the same intended system.

In other words, intelligence needs scaffolding.

This is the emerging role of the AI software factory: not simply using artificial intelligence to generate pieces of code, but constructing workflows around that intelligence that allow it to repeatedly apply different kinds of expertise toward a larger, coherent outcome.

At the simplest level, AI helps a programmer write a function.

Then it helps build a feature.

Then an application.

As the surrounding workflow becomes more capable, however, the scale of the artifact that can be produced begins to expand.

A human can describe an idea conversationally. AI can help organize that thinking into requirements. Requirements can become architecture. Architecture can become implementation tasks. Those tasks can invoke different languages and technical capabilities. The resulting components can be assembled, tested, corrected, deployed, and iterated.

We are moving toward a point where a person who understands a system at the architectural level may no longer need to possess expert mastery of every technical language required to construct it.

Instead, that person needs to understand the system well enough to direct the intelligence that does.

They need to understand the major layers, relationships, dependencies, behaviors, constraints, and desired outcome. They need enough architectural literacy to know what should exist and how the pieces should relate. They then translate that understanding into whatever form the software factory requires at its front door: conversation, models, requirements, product documents, architecture specifications, workflows, or some combination of them.

The factory can increasingly translate that intent downward into specialized implementation.

Python where Python is appropriate.

Swift where Swift is appropriate.

Databases, infrastructure, APIs, testing, security, deployment, and other capabilities where they are required.

The human no longer has to become the deepest practitioner in every one of those disciplines before being capable of directing their use.

Build Bigger, Stay Valuable lays out why I think this changes the size of the thing one person can be responsible for. What follows here is what that means for how the work actually gets architected.

Why Constellation Programming Now

AI changes not just coding speed, but the economics around which software engineering was organized. Historically, implementation was expensive, so rational engineering practice was to build only the needed capability, avoid unused infrastructure, and expand only as requirements clarified. Those were not bad practices — they were rational responses to expensive implementation. AI-native factories lower that cost. What follows here is the practical consequence for how software actually gets architected.

The Unit of Economic Value Gets Larger

- Existing environments typically have many applications, each built one funded requirement at a time — a "technological Frankenstein." That is not a failure of the people who built them. It is the rational accumulated result of solving one funded problem at a time under expensive-implementation economics.

Stop Starting With the Application

- Traditional requests start with an application: a volunteer app, a transportation system, a registration app.
- Constellation Programming begins instead by choosing the ecosystem.
- Ask: who and what exist? What do they do?
- City primitives: people, organizations, locations, facilities, vehicles, properties, equipment.
- Activities and transactions: register, transport, employ, host, request, inspect, pay, attend.
- This is observation, investigation, and domain understanding — architecture before programming.
- The goal is to identify the durable primitives, along with their relationships, activities, and transactions.

Establish the Rules Before Pouring the Foundation

- Architectural rules must be set first: identity, authorization and visibility, provenance, time and change, tenant isolation, information movement, model evolution and compatibility, and handling of sensitive information.
- These choices have direct schema consequences — for example, what provenance metadata is captured, or how a child's data and access are governed.
- Only then does the schema get built: primitives, collections, relationships, predicates, attributes, transactions.
- From the schema, the factory builds create/retrieve/update/manage machinery, then services, then applications and interfaces.
- Simplified sequence: understand the ecosystem → identify what exists → identify what happens → establish architectural rules → model the schema → build the foundational machinery → build services → build applications and experiences.

The House We Always Wanted to Build

- A couple builds a house for two, but the future may include kids, a parent, a home office, a workshop, an extra bathroom, or an apartment.
- Under ordinary economics, it would be absurd to build every future room and all the infrastructure for it up front.
- Later additions are expensive: excavation, foundation, plumbing, electrical, walls.
- Growth means both more rooms and more capable rooms.
- If foundation, electrical, plumbing, and structural preparation become almost free, the rational decision changes.
- The point is not to build every future room, but to prepare the property.
- AI factories create an analogous architecture economics for software.

AI Did Not Discover Better Architecture

- Engineers have always known that coherent architecture is good practice.

- Thoroughness simply produced enormous implementation work: schema, validation, CRUD handlers, APIs, permissions, tests, documentation, integration, maintenance.
- So teams combined concepts, deferred infrastructure, and built only the first application's model.
- AI changes the economics of laying the "foundation bricks."
- The human does the consequential intellectual modeling; the factory generates the repetitive machinery.
- The marginal cost of architectural precision falls.
- The right question becomes "Is this actually a distinct thing in the world?" rather than "Can we afford another collection?"
- AI did not discover good foundation practice — it removed the constraint that was preventing it.
- The same technology that makes small, one-off software less economically significant is what makes large, extensible systems achievable.

What Constellation Programming Requires From Engineers

- The factory handles implementation; the human is responsible for the more consequential upstream decisions.
- The factory can create what it is asked to create — someone still has to understand what ought to exist.
- That requires systems literacy across compute, storage, I/O, networks, databases, identity, security, distributed systems, cloud, data architecture, services, interfaces, and applications.
- Not mastery of every layer as a specialist — architectural literacy across all of them.

Why a New Curriculum Is Necessary

- Current practices were shaped by previous economics, and appropriately so.
- New economics require reconsidering what the optimal practices are.
- This requires more than AI coding tools or knowing how to prompt a factory.

- It requires a methodology for deciding what the factory should build: investigating the ecosystem; recognizing durable primitives; separating what exists from what happens; reasoning about identity, provenance, time, security, communication, and change; building schemas for capabilities that haven't been requested yet; distinguishing productive foundation work from speculative feature development; supporting new services as well as more capable existing ones; and understanding the layers of computing beneath it all.
- The purpose of Constellation Programming is not simply to use AI to build more software. It is the engineering discipline needed once implementation is cheap enough that the limiting factor moves upstream — to understanding, architecting, and directing the systems worth constructing.
- The factory changes what we can afford to build. Constellation Programming addresses what we now need to learn in order to build it well.