

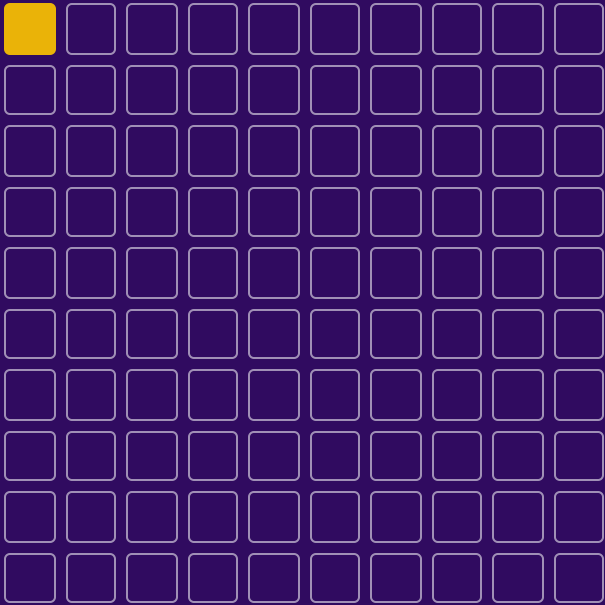
FOR ENGINEERING AND BUSINESS LEADERS

We Will Write 100× More Software in the Future

Software is about to escape the software industry.

INSIDE

- More problems than software
- The new economics of software
- Every problem as a simulation
- Disposable software
- Continuous experimentation
- From writing code to grading work
- Why 100× is plausible
- Physical AI
- A more computable world



■ SOFTWARE TODAY THE SOFTWARE TO COME

INTRODUCTION

Software is about to escape the software industry.

Software reshaped finance, media, retail, and transportation over the last few decades. Walk into a factory, a farm, or a hospital, though, and most of the work still runs on spreadsheets, phone calls, and people who have done the job for twenty years.

INDUSTRY	WHAT IT RUNS ON TODAY
Factories	Spreadsheets and human judgment to schedule production
Farms	Experience and limited data
Hospitals	Fragmented systems and manual workflows
Construction	Documents, phone calls, and project managers
Supply chains	Rules designed years ago and rarely questioned

These industries have plenty of problems software could solve. The software was just too expensive to build and keep up for each of them. AI changes that, and the effect goes well beyond making developers more productive.

100x

We think the world will write 100 times more software. Writing code will get faster, but the bigger reason is that a whole universe of new problems will become worth solving with software.

HERE IS WHY

- 01

More problems than software
- 02

The new economics
- 03

Software from outside software
- 04

Every problem a simulation
- 05

Disposable software
- 06

Continuous experimentation
- 07

More specialized software
- 08

Grading the work
- 09

Why 100x is plausible
- 10

Physical AI
- 11

A more computable world

Software has only been built for problems big enough to sell.

The industry grew up around problems big and common enough to sell to thousands of customers. A vendor builds one product, everyone configures it a little, and the edge cases get handled by hand. That model works, but it means the problems specific to one plant, one route network, or one lab rarely get software of their own.



The vendor model. A problem has to be common enough to justify the cost of building for it.

Where the line falls today

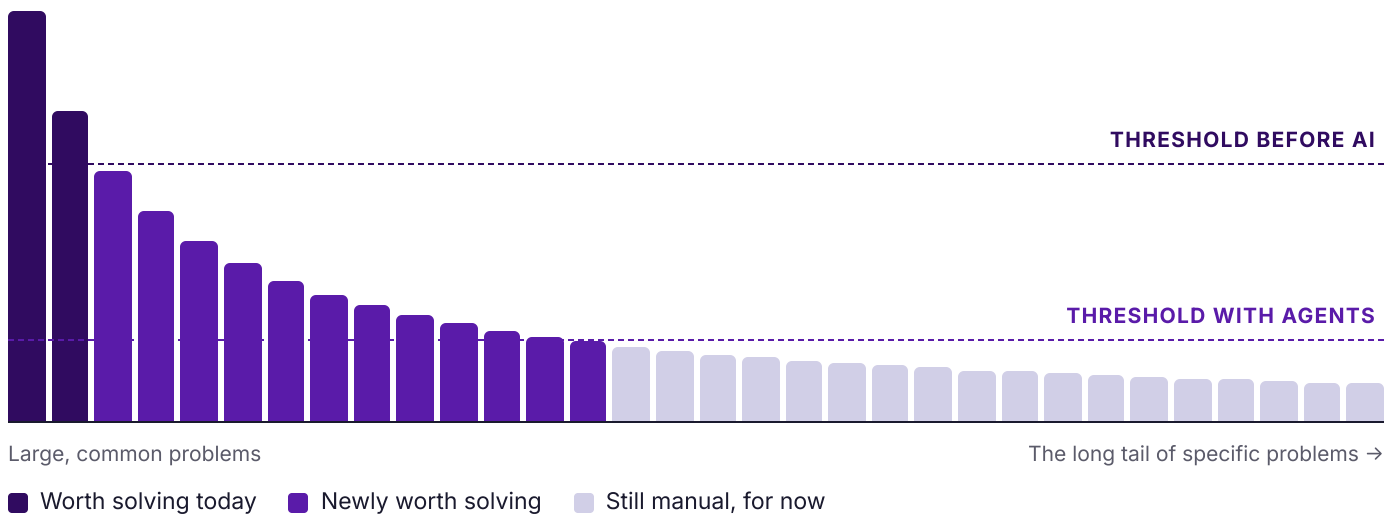
Factory	An unusual mix of machines, suppliers, maintenance schedules, and production constraints.	SOFTWARE The standard scheduling features	PEOPLE The exceptions, every shift
Logistics	Routing that depends on local traffic, vehicle capacity, delivery windows, weather, and customer rules.	SOFTWARE The common routing case	PEOPLE Every route that doesn't fit
Retail	Inventory allocation that changes with every season, location, supplier, and promotion.	SOFTWARE Basic replenishment	PEOPLE Each season's allocation calls

None of these are unsolvable. The cost of building the software has simply been higher than the benefit anyone could see.

There is no shortage of problems software could solve. What has been missing is a cost of building software low enough to make solving them worth it.

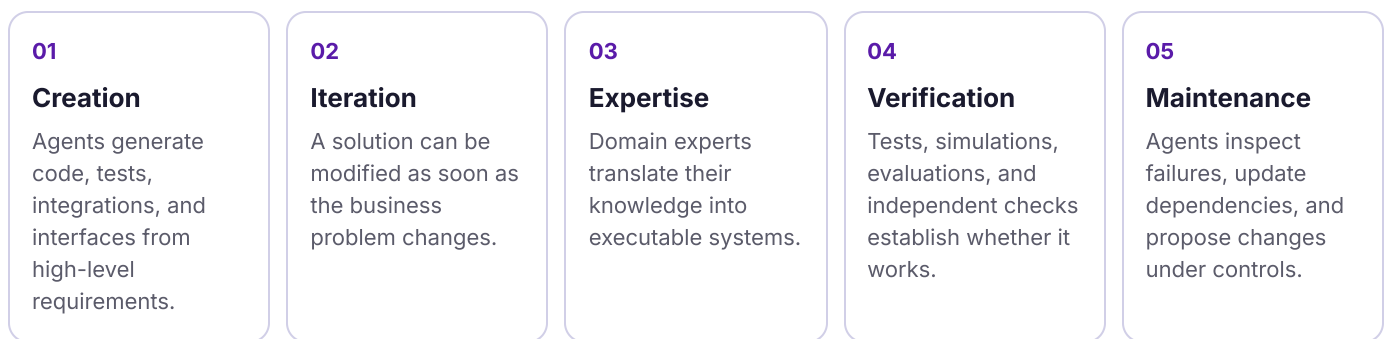
Smaller problems are now worth writing software for.

A small internal tool can still take a few weeks of engineering time, and that is before anyone has to maintain it. Processes change, an API gets deprecated, a dependency needs upgrading. Add it all up and plenty of important problems never make it onto a roadmap. AI brings down both the cost of building and the cost of keeping things running, so the bar for what is worth building drops.



Each bar is a problem, sized by the value of solving it. Lowering the cost of software moves the line.

Where the cost comes down

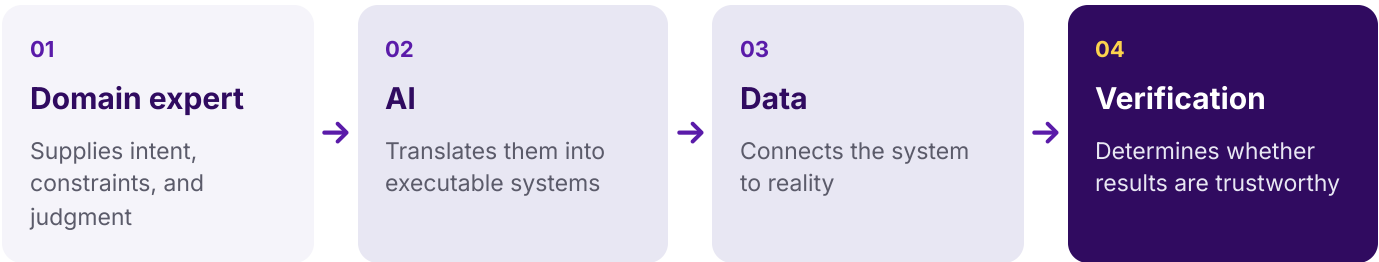


IN PRACTICE

A tool that once needed three engineers for a month might now take one engineer working with agents far less time. Gains vary a lot by task, and requirements, integration, and correctness still take real effort.

The next wave of software will be written by people who don't work in software.

We usually picture software being built inside tech companies and engineering departments. That picture is starting to look dated. A manufacturing engineer, a hospital operations lead, or a grower already knows the constraints of their work better than any vendor. With agents doing most of the translation into code, they can build tools for their own problems without becoming developers.



Who does what when domain experts create software.

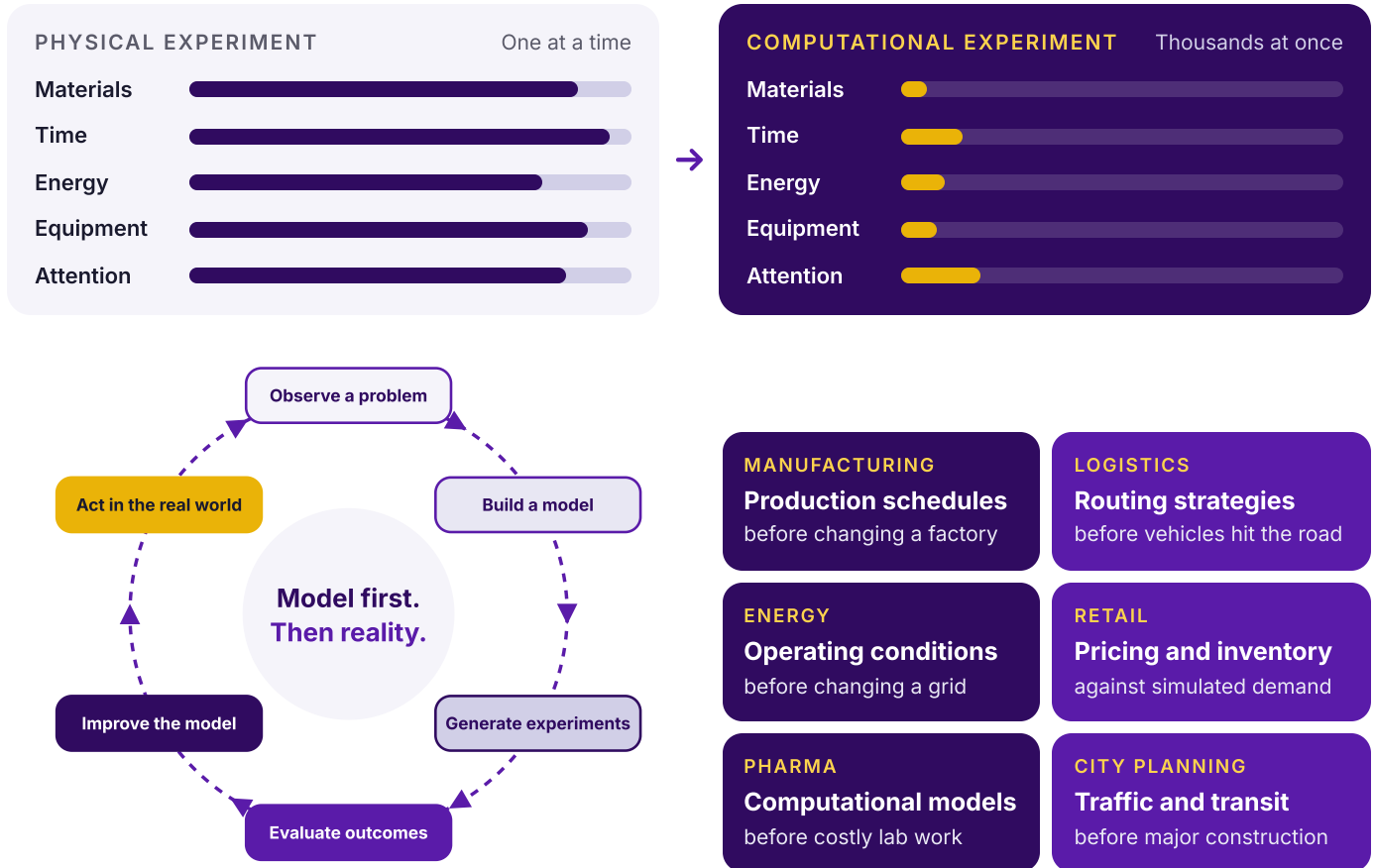
What it looks like

Manufacturing	An engineer describes the constraints and production data. An agent builds a simulation, tests it against history, and runs thousands of schedules against throughput, energy, delivery, and machine availability.
Hospitals	An operations team models patient flow, staffing, rooms, and appointments, and tests staffing policies before changing the real world.
Agriculture	A grower generates software for a specific mix of soil, weather, irrigation, crop variety, and local economics, updated as new data arrives.

As that happens, far more people and organizations will be able to build software that is genuinely useful to them.

Try it in a model before you try it for real.

A lot of operational improvement still happens by trial and error on the real thing, plus experience, manual analysis, and a handful of experiments. That is slow and expensive, and sometimes dangerous. When a model of the problem is cheap to build, you can run hundreds of experiments in it and take only the promising ones into the physical world.



The feedback loop. The more accurate and easy to build models get, the more decisions can go through it.

Every model is software too. So demand for software grows with how deeply we study each problem, not just how many problems there are.

Ingestion pipelines Data transformations Optimization algorithms Evaluation systems Interfaces

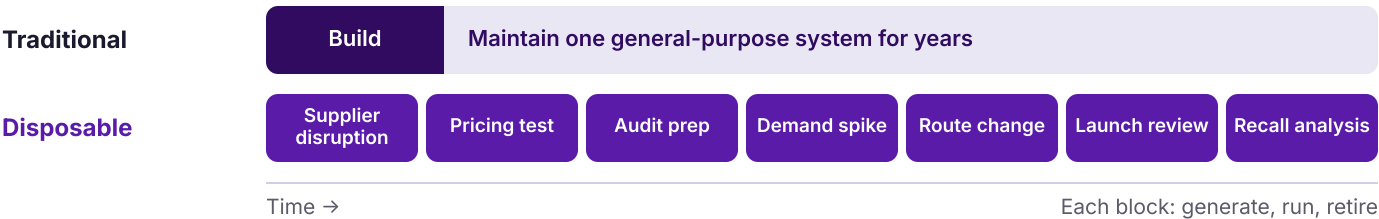
Integrations

A CAVEAT

A model is not reality. A confident answer from an inaccurate model can be worse than no answer at all, so assumptions, data, and validation matter.

Some software only needs to exist for a week.

Building an application has always been a commitment. You pay for the build, then for years of maintenance, so it makes sense to build general systems that serve many uses for a long time. When generating an application costs little, you can build one for a single problem, use it while the problem lasts, and then retire it.



Same period, different lifecycle. Each purpose-built tool lives only as long as its problem.

EXAMPLE · SUPPLIER DISRUPTION

An agent generates a temporary application combining purchasing history, supplier lead times, inventory, contracts, and demand forecasts. It runs recovery strategies and recommends a plan. Once the disruption is resolved, the application is no longer needed.

Disposable doesn't mean careless

It refers to how long the tool lives and what it costs. It still has to be correct for the job it is doing.

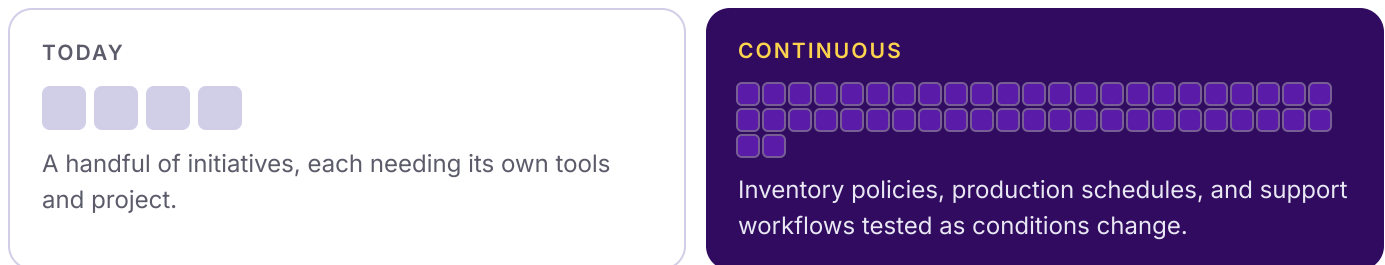
We already do this

Most teams already rely on throwaway spreadsheets, scripts, and prototypes. AI makes it possible to do the same with full applications.

When a purpose-built tool is that cheap, a generic one that only half fits gets harder to justify.

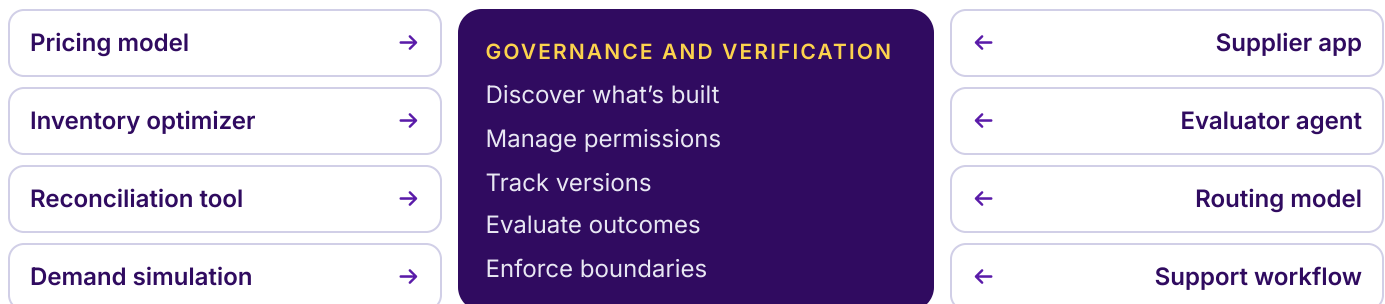
Experiments stop being quarterly projects.

Most companies run only a few serious experiments a quarter, because each one needs engineering time and its own tooling. If agents can build that tooling on demand, they can also keep proposing changes, testing them, and handing the evidence to a person to decide. Experimentation turns into something that runs in the background all the time.



The limit becomes how many results we can evaluate reliably, rather than how many experiments people can organize. Changes with real impact should still need approval and safeguards.

Lots of small systems need a way to be governed



More software means more interfaces, more dependencies, and more ways for things to fail. Keeping track of what exists, who can run it, and whether it works will matter as much as building it.

The hard part becomes checking the work.

Agents can already write more code than any team can read line by line. So the question shifts from who writes the code to how we know it is right. Unit tests still help, but most business problems have to be judged on outcomes: cost, reliability, safety, and whether the original problem actually got solved.

Questions every team will need to answer

- 01 Does the generated application actually solve the original problem?
- 02 Does the simulation predict real-world outcomes?
- 03 Does the optimization respect all constraints?
- 04 Does the new version improve performance without regressions?
- 05 Is the agent operating within its permissions?

Matching the check to the risk

Precise checks

Tests, formal constraints,
reproducible experiments

Correctness can be checked exactly

Evidence

Simulations, historical comparisons,
independent evaluators

Judgment

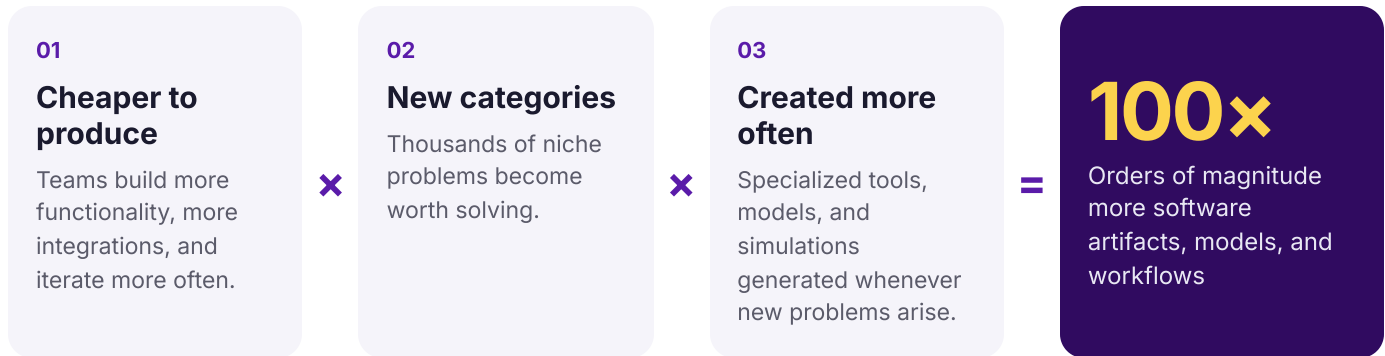
Expert review and observed real-
world results

Correctness can only be estimated →

Getting this right is what lets a company build more software without also shipping more failures.

Where the 100× comes from

100× is a thesis, not a forecast anyone can measure today. We are not saying each developer will write 100 times more code. We are talking about how much software the economy as a whole can justify building once three things happen together.



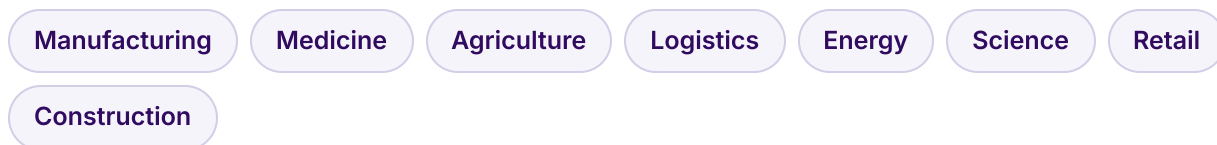
The first grows software for existing needs. The second expands the needs software can address. The third raises how often solutions are created and replaced.

WHAT COULD SLOW IT

Adoption will be uneven. Some software will stay expensive because the hardest parts are data access, safety, integration, and verification, not code generation.

Where most of it will show up

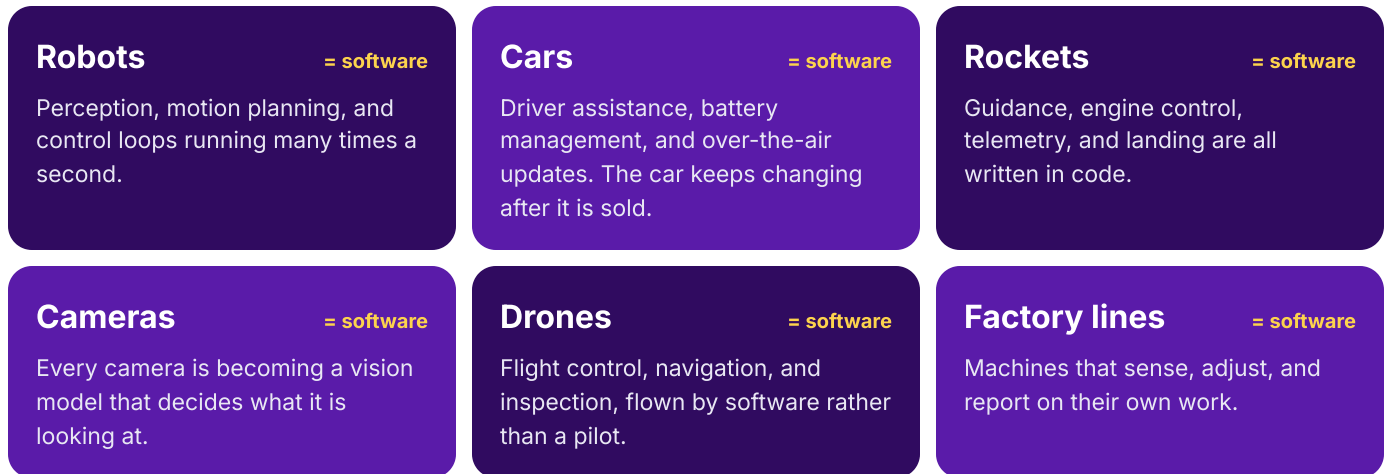
Most of this growth probably won't show up in software industry numbers. A lot of it will be built inside manufacturers, hospitals, farms, and utilities, and never sold or downloaded as a product.



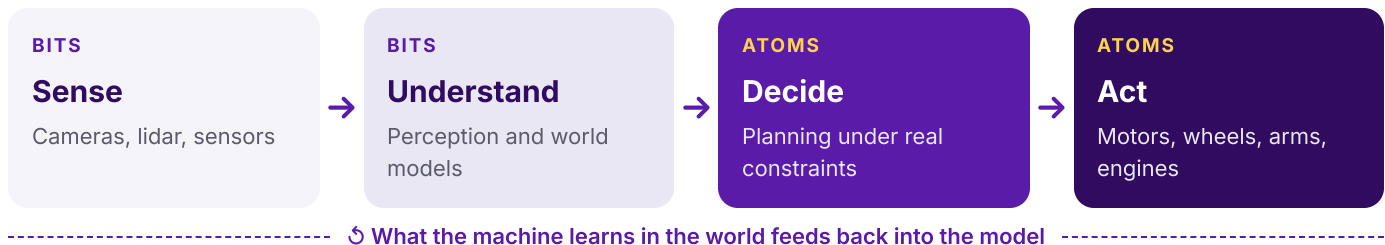
We will write 100× more software because we will find 100× more reasons to write it.

When bits meet atoms, software multiplies.

A robot is software with a body. So is a modern car, and so is a rocket. Every camera is turning into a model that understands what it sees. As AI gets good enough to see, decide, and act in the physical world, each new machine brings a lot of software with it, and most of that software doesn't exist yet.



The loop every physical AI system runs



One machine, many systems

Around every robot or vehicle sits the software to train it, test it in simulation, prove it is safe, update it in the field, and watch the fleet.

Firmware

Perception models

Simulation

Safety verification

Fleet updates

Data pipelines

Digital twins

Monitoring

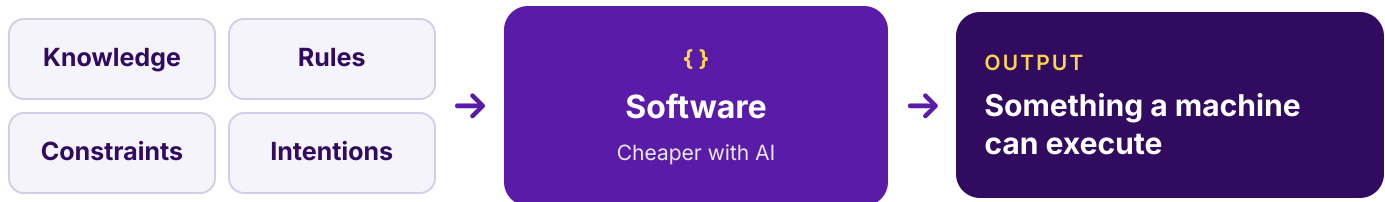
HIGHER STAKES

A bug in a spreadsheet costs time. A bug in a robot arm or a braking system can hurt someone. Physical AI raises the bar for simulation and verification even further.

Every machine that learns to see and move will need software to do it. **That is where a large share of the next 100× will come from.**

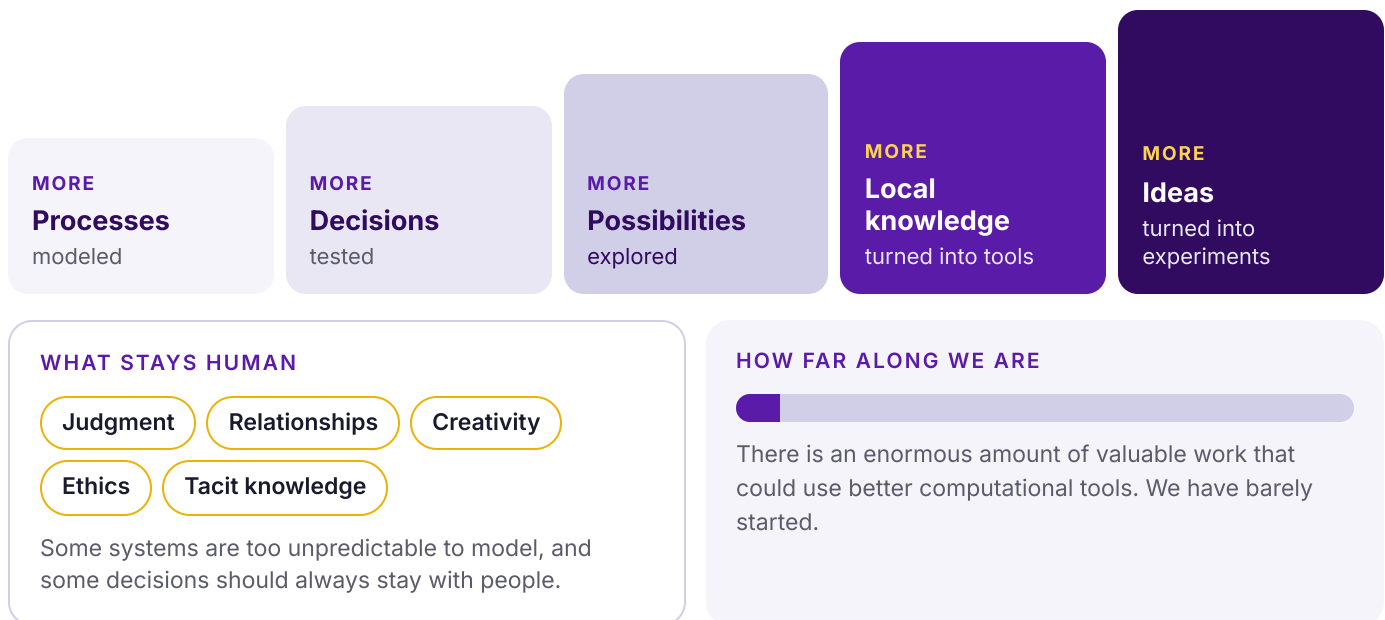
More of the world becomes computable.

Writing software means taking some piece of how the world works and describing it precisely enough for a machine to run. We have always been selective about which pieces, because it was expensive. AI changes how much of the world we can afford to describe this way.



What writing software actually does.

As the cost falls, more becomes possible



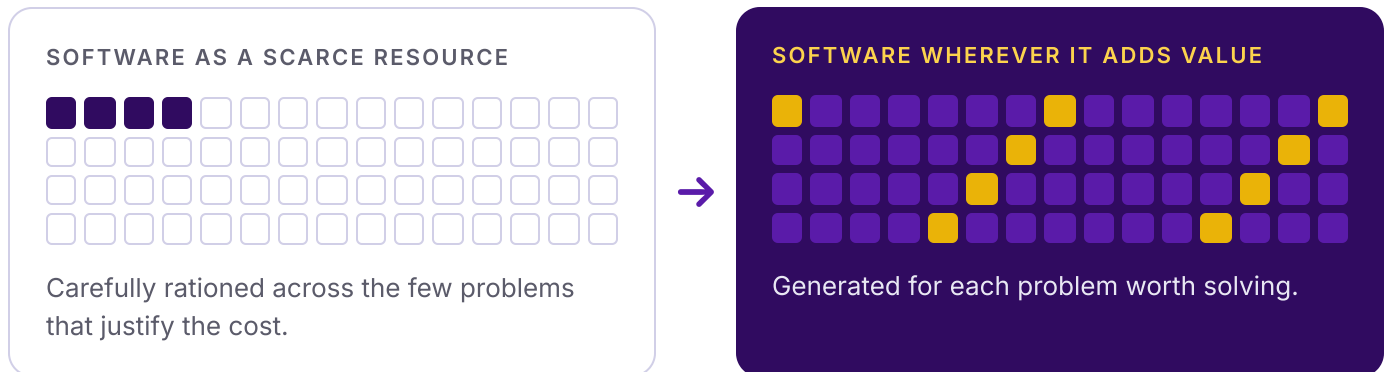
The next big wave of software probably won't come from making existing software a little better. **It will come from bringing computation to places where it was never affordable.**

From building software faster to asking a different question.

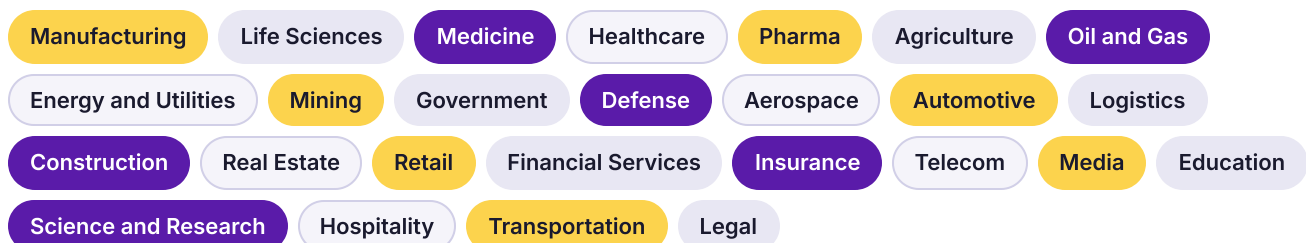
For years the software industry has focused on helping people build software faster. AI does that, and it also changes what we should be asking.

- 01 What happens when building a purpose-built tool costs little compared to the problem it solves?
- 02 What happens when every business process can have its own model, and every hypothesis can become an experiment?
- 03 What happens when software is created on demand, run continuously, checked independently, and thrown away when something better comes along?

From scarce to abundant



Not just more software companies or more programmers. More software in:



WHAT COMES NEXT

100x more software isn't far-fetched.

The pieces are already here: models that write code, agents that test and fix it, and machines that turn software into action in the physical world. What's left is deciding what to build.

**Every company on the planet will
become a software company.**
Imagination is the only bottleneck.

01 Built by people

who never thought of themselves as
developers

02 Built for a day

and retired once the problem is gone

03 Built to experiment

at a scale no team could organize by hand

04 Built to model the world

so decisions are tested before they are
made

SOFTWARE BECOMES

Disposable

AI BECOMES

Leverage

EXPERIMENTATION RUNS

24/7/365

**Planning for 100x more software? Talk to the
XHawk team.**

AI Employees for Product, Engineering & Analytics.

xhawk.ai →