Interoperability in computational economics

Towards a shared (modular) computational representation for economic models

Chris Carroll & **Akshay Shanker** (Econ-ARK) · **Matt McKay** (QuantEcon)

CEF 2026 · Venice

Opening talk of the session. Three parts of roughly twenty minutes: the diagnosis, the precedent, the program.

Roadmap

Part 1 · The case for interoperability — *the diagnosis.* Economists share ideas freely, yet rarely share working code; we show the gap and what it costs.

Part 2 · Lessons from other fields — *the precedent.* Compilers, machine learning, and databases met the same problem and answered it with a shared representation.

Part 3 · Our work — *the program.* A common representation for dynamic models, pursued on two tracks:

- the **Bellman calculus** — *Project Bellman*, with Econ-ARK;
- a **model repository** of reusable components, with QuantEcon.

Part 1 — The case for interoperability

Economic models share structure, yet our software usually does not represent that structure

> The problem is not poor code. The problem is that **model meaning lives in people, not explicitly in software**.

> **Cost:** limited inter and intra-temporal spillovers. *All* our marginal returns can be higher.

> **Aim:** Solve using existing **modular software** where possible. Build new models by recombining existing **operators** — reusing or swapping their numerical methods — and adding new modular **operators** only when needed.

Start from the object we all recognise

A large and important class of dynamic models share a common structure given by the recursive Bellman equation:

$$V(m) = \max_{0 \leq c \leq m} \left\{ u(c) + \beta \, \mathbb{E}_{y'} \left[V\left(R(m-c) + y'\right) \right] \right\}.$$

> The Bellman equation has a clear **operator** structure, yet implementation usually hides the operators

The Bellman operator splits into named pieces

Computing today's value runs these operations **backward** — from next period's value to this one (order depends on shock timing; here the shock is post-decision):

The idea: each box is a separate, reusable piece — a method you could borrow from a shared library and swap (a quadrature, an interpolation, an optimiser) without touching the others.

> **What is done today:** the boxes are hand-written and fused inside one loop — so you cannot borrow or swap just one.

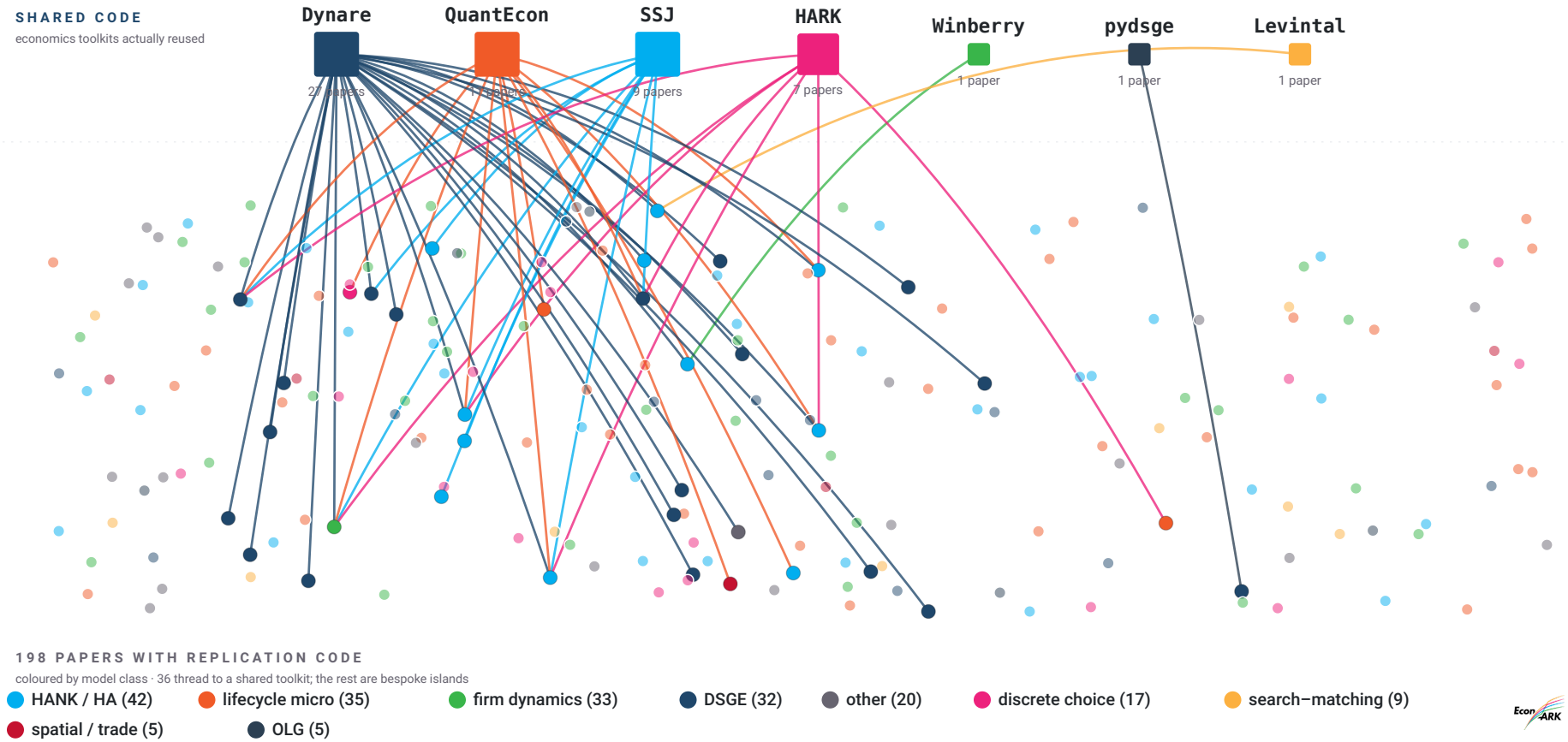
One mathematical object, many code paths

Recursive problems are solved in many capable toolkits:

- **Dolo** — model files, global solution methods, simulation;
- **Dynare** — model files, perturbation and local methods, simulation;
- **HARK** — lifecycle solvers, interpolation, simulation;
- **QuantEcon** — dynamic programming routines and numerical primitives;
- **ConSav** — consumption–saving and EGM-style routines;
- **SSJ** — sequence-space methods for heterogeneous-agent models.

> The gap is that there is no common calculus that allows the representations to **talk to each other** and to the economic model itself — not even where **Dynare** already standardises model files within one paradigm.

The empirical picture — ideas connect, code does not

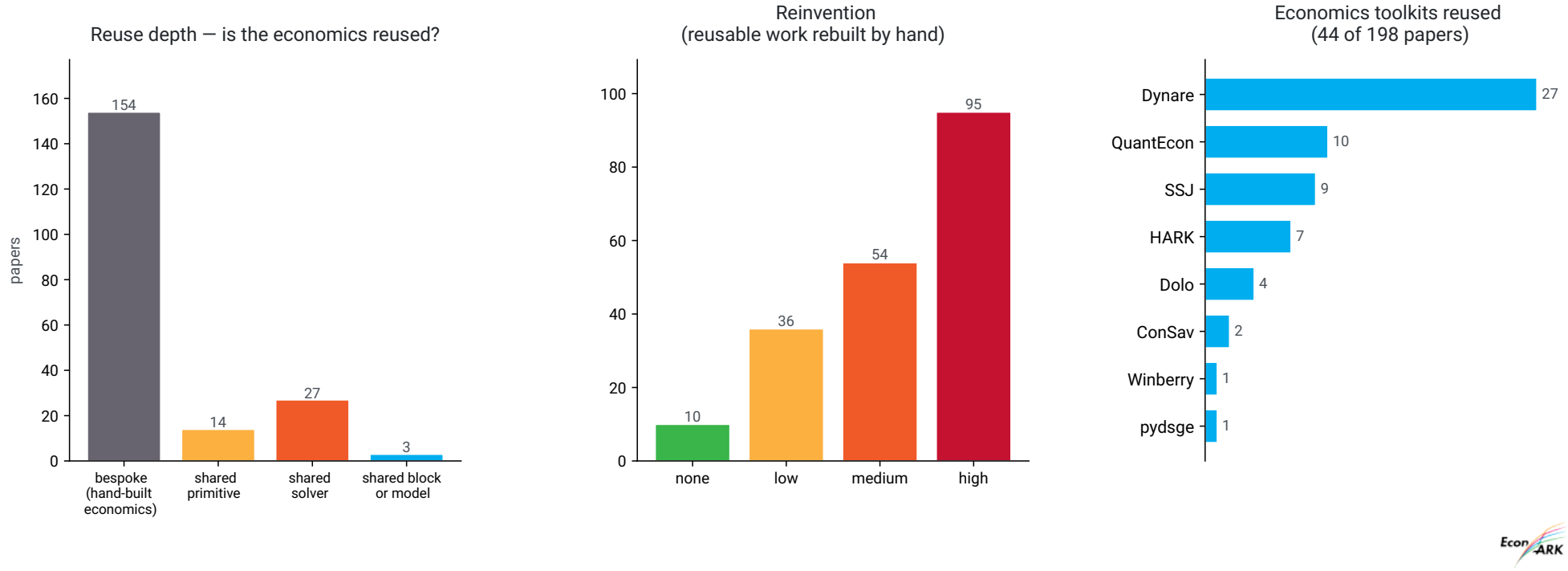


Papers lean on the same handful of ideas — **Aiyagari–Huggett**, **Krusell–Smith**, **Rust** — yet the code almost never travels. Of 198 papers with replication code, only **36 reuse a shared economics toolkit**; the rest are bespoke islands.

How much code is actually shared?

Code reuse across recent dynamic-economics papers

500 papers screened across the top ten journals (2023-26) · 257 computational · 198 with replication code. Reuse depth and reinvention judged against the capability catalogue.



Reuse depth — does the code reuse the *economics*, or rebuild it? **Bespoke** (hand-built, even on top of `numpy` / `scipy`) → shared **primitive** → **solver** → **model block**. **Reinvention** — how much reusable functionality each paper rebuilt by hand.

500 papers screened across ten leading journals (2023–26), 198 with replication code. **154 build the model's operations by hand** — bespoke is not always wrong, but economics rarely travels as reusable modules — while only 44 (≈22%) reuse an economics toolkit at any depth.

Why does this happen?

1

Procedural code

The solver loop becomes the model representation.

2

Mathematical model vs comp. method

Economic assumptions and numerical choices get bundled.

3

No common calculus

No semantic (meaning) representation: timing, states, shocks, and operators remain implicit.

4

Incentives

Papers reward results, not portable model objects.

We focus on the first three. The incentive problem belongs in the working-group discussion.

Procedural code is excellent for execution

A solver must choose an order of operations:

```
for i, m in enumerate(m_grid):
    vals = []
    for c in c_grid[c_grid ⩽ m]:
        a = m - c
        EV = expected_value(V_next, R*a + y_nodes, weights)
        vals.append(u(c) + beta * EV)
    V[i] = max(vals)
    c_policy[i] = c_grid[argmax(vals)]
```

This is useful code: it makes the computer solve the problem.

> But the important object is hidden behind the innocent-looking name:
>
> `expected_value(...)`

PROBLEM 1 · PROCEDURAL CODE

One name, two different objects

What the equation says

$$V(\cdot) \;=\; \beta\, \mathbb{E}_{y_\succ}\left[\, V_\succ(R\,\cdot + y_\succ)\,\right]$$

An operator: it takes the continuation value **function** $V_\succ$ and returns the value **function** V — without fixing **how** the expectation is computed.

What the code usually contains

```
def expected_value(V_next, m_next, weights):
    vals = interp(V_next, m_next)
    return np.dot(weights, vals)
```

This is not just “the expectation.”

It is a **procedural numerical routine** — one interpolation rule, one grid convention, one quadrature, one array layout, one treatment of off-grid states.

> So the procedural problem is not the outer loop alone.
> It is that a mathematical operator like $\mathbb{E}$ gets replaced by a piece of procedural code whose numerical assumptions are built into its body.

Changing the model representation should mean swapping an expectation operator.
In procedural code, it often means opening `expected_value(...)` and rewriting the numerical machinery by hand.

PROBLEM 1 · PROCEDURAL CODE

The hard-coding problem

Procedural implementations often bake numerical assumptions into the control flow:

CODE OBJECT	WHAT IT MAY HARD-CODE
`m_grid` , `c_grid`	state and choice discretisation
`y_nodes` , `weights`	shock approximation and integration rule
`expected_value(...)`	expectation, interpolation, and continuation-value evaluation
`for` loop order	update schedule and memory layout
`argmax(vals)`	optimization method and admissible choice set

> Changing the method then becomes **code surgery**, not a model-preserving substitution.

Four layers are tangled here that should travel **separately** — the economic model, the timing convention, the numerical method, and the implementation.

PROBLEM 1 · PROCEDURAL CODE

Backus (1978), translated into economist language

Backus's point for us is not “use a different programming language.”

It is this (Backus, *Can Programming Be Liberated from the von Neumann Style?*, CACM 1978):

> *Step-by-step code can compute an object while hiding the operations (operator algebra) that define it.*

For Bellman problems, the algebra is familiar:

$$\text{transition} \quad \mathbb{E} \quad \text{reward} \quad \max \quad \text{value}.$$

> Economic models become portable when value functions and distributions are **named mathematical objects** you can pass around and swap — **first-class objects** in the programming sense — by changing methods and settings, not just steps inside a loop.

A second toolkit can then reuse them directly, instead of re-deriving the hidden model by hand.

PROBLEM 2 · MODEL VS COMPUTATIONAL METHOD

Economics and computation get blurred

A program discretises the model onto a finite grid:

$$P_N \text{ on a finite grid}, \qquad \hat{\mu}' = \Gamma_N(\hat{\mu}, \pi_N).$$

Is that **the economic model** — a world of isolated grid points, some with zero income — or a **numerical approximation** of the continuous law of motion?

$$z' \sim P(z, \cdot), \qquad \mu' = \Gamma(\mu, \pi).$$

It is the approximation: $\Gamma_N \to \Gamma$ as the grid refines — a **method**, not a model.

> When the discretisation is mistaken for the economics, the **same model** under two grids reads as **two different models** — and their code never meets.

PROBLEM 3 · NO SHARED SEMANTICS

One shock, two hand-coded conventions

A paper says: *"a policy shock raises entrants at date t."* The sentence does not pin down the model — the code fixes the timing **by hand**, in bespoke conventions and code comments.

POLICY SHOCK AT T	
CONVENTION A	**CONVENTION B**
The shock hits the distribution at date t.	The shock hits the distribution at t + 1.
No transition from the entering-period distribution.	A transition from the beginning-of-period to an intermediate (mid-period) distribution.
Time-t prices respond.	Time-(t + 1) prices respond.

> Each convention lives in **hand-written code and comments**, not in a **shared within-period stage** (cf. Auclert–Bardóczy–Rognlie–Straub, SSJ, *Econometrica* 2021). So a module built for one cannot be reused in the other without hacking the hand-coded timing — and the impulse response differs before any grid or solver is chosen.

PROBLEM 3 · NO SHARED SEMANTICS

Even a familiar refactoring blocks reuse

The same problem, written from the **same primitive operators** two ways:

Primitive form — shock after the choice; expectation inside the max:

$$V(m) = \max_{c} \left\{ u(c) + \beta \, \mathbb{E}_{y'} \, V(R(m-c) + y') \right\}$$

Factored form — expectation pre-computed into a continuation value W:

$$W(a) = \beta \, \mathbb{E}_{y'} \, V(Ra + y'), \qquad V(m) = \max_{c} \left\{ u(c) + W(m-c) \right\}$$

> We are **not** arguing for one factorisation over the other — both are standard (Carroll 2006; Sargent–Stachurski). The point is that they must **relate at a higher level**: a common calculus where a package written in primitive form *can* supply the continuation-value operator, instead of W being rebuilt by hand.

The cost of hand translation: a summary

Three problems compound into one cost:

1. **Procedural code** — the solver loop becomes the model representation.
2. **Model vs method** — economic assumptions and numerical choices are bundled together.
3. **No shared semantics** — timing, states, shocks, and operators stay implicit.

So each new large structural model is built as a **monolithic code block, from scratch**, and the reusable unit stays the **paper-and-repo pair** — a coauthor re-solving in another toolkit starts from the appendix, not a portable model object.

A model calculus: shared semantics, then modular reuse

A **model calculus** is a *semantics* for models: it fixes the **meaning** (the economics) — states, choices, transitions, timing — while the backend chooses the **execution** (the numerics), as **SQL** does vs. its query planner.

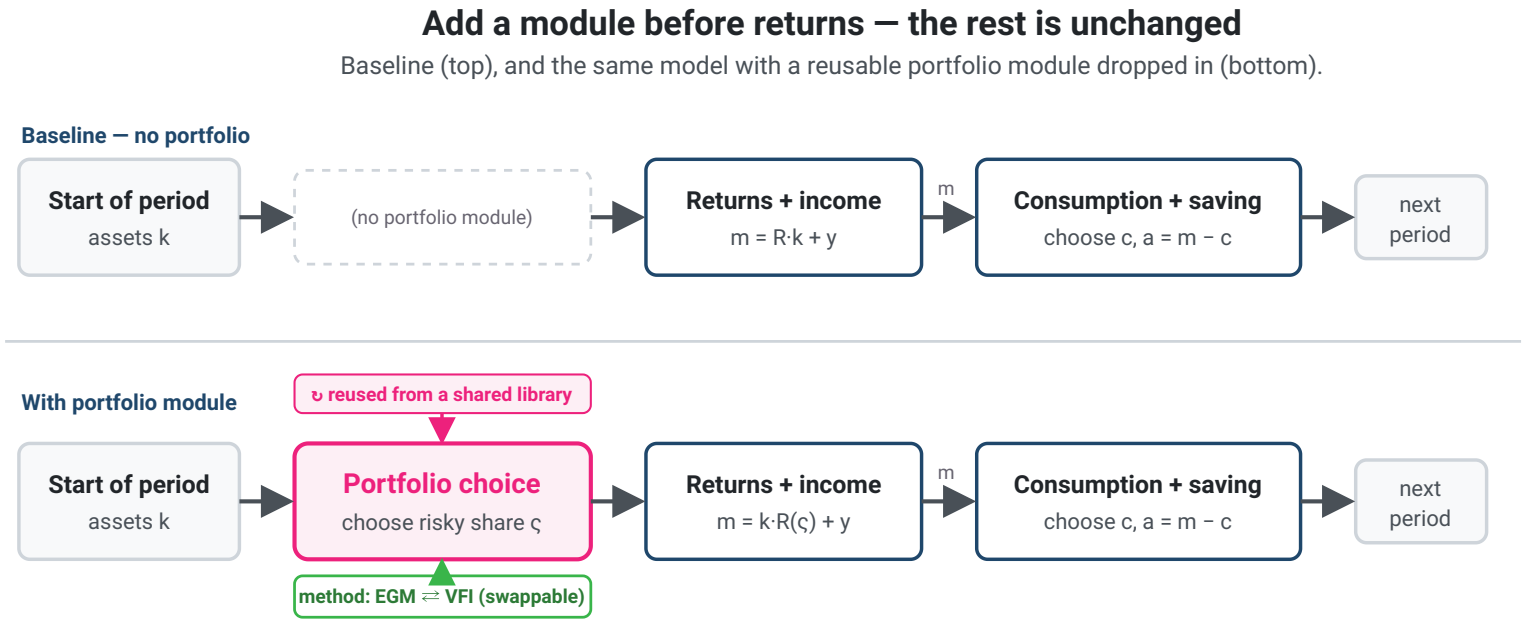
> This is **semantics, not syntax**: it pins down *what a model means*, not *how it is written*. Two files in different notation that denote the same object are the **same model**.

With meaning fixed, the reusable objects are **named modules** — a transition, an expectation, a reward, a one-period update — each small enough to reuse, not a whole solver. This lets us:

- **swap the method** for one operation without changing the model (within its structural domain — EGM needs invertibility, etc.);
- **reuse** another author's transition or update, with timing made explicit;
- **solve one model with several backends** — cooperating toolkits, same model;
- let **AI translate the plumbing** while **model meaning stays fixed**.

A reusable module, dropped in before returns

Drop in a **portfolio-choice module** before returns — reused from a shared library, with a swappable numerical method. The cash-on-hand hand-off and everything after it stay fixed.



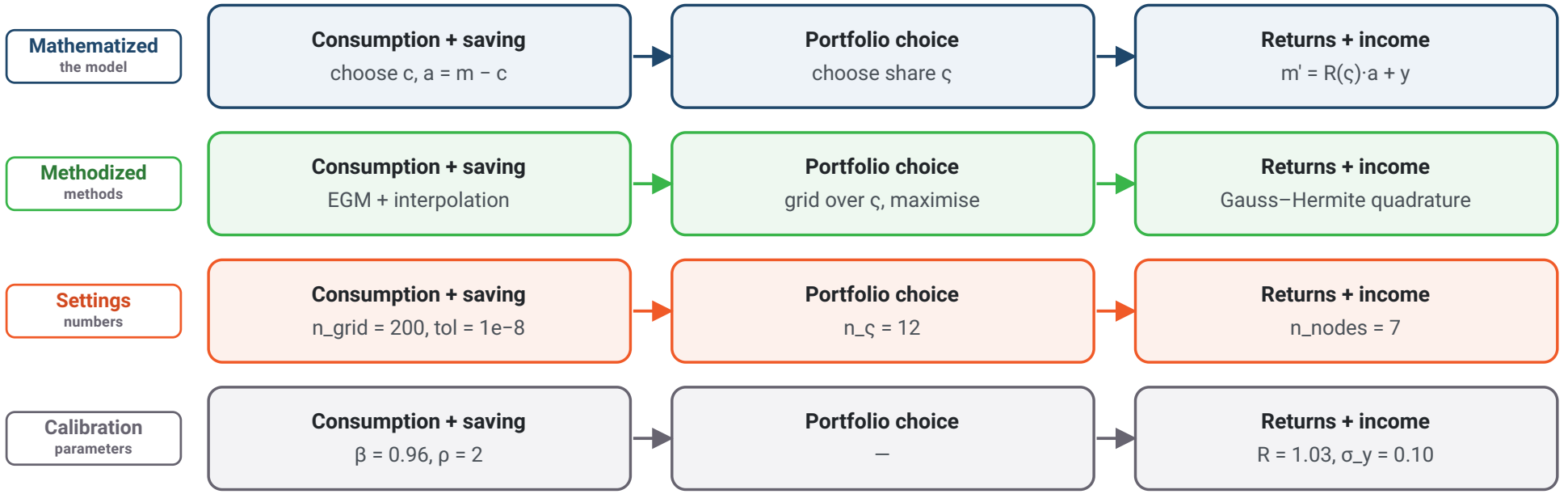
Drop the reusable module into the slot before returns — returns, consumption, and everything after stay fixed.

Methods and settings layer onto a fixed structure

The economic structure is fixed. The numerical method, the settings, and the calibration sit on top as separate layers — each chosen, and swapped, independently.

One structure, four layers

The same three stages in every row — the boxes never move. Only the layer changes: method, settings, calibration.



Each row is the same pipeline. The boxes never move — only what lives in them changes, layer by layer.

The scope of the claim

In scope now

Individual Bellman problems — lifecycle and infinite-horizon consumption–saving, and portfolio and consumption variants.

Close cousins

Estimation and simulation around solved models, mean-field formulations built from Bellman problems, and backend-to-backend robustness checks.

The ambition

Production-side and firm dynamics, and CGE / general-equilibrium models composed from the same blocks.

Not claimed here

Game-theoretic equilibrium concepts, and universal coverage of all economic models.

The honest claim is narrower than "all economic models" — but Bellman problems and their close cousins are already large enough to matter, and structured enough to make interoperability practical.

Is this possible?

Three reasons to think yes:

- some economists already compose models from reusable blocks — the **sequence-space-Jacobian** community builds HANK models from `het_block` and `simple_block` pieces (our census found SSJ reused inline in several papers);
- the mappings between toolkits already exist in papers, private notes, and translators' heads;
- other fields hit the same wall and built shared representations that work.

> Parts 2 and 3 ask how other fields did it, and how we build the corresponding program for economic models.