AI & ML for Quant Macro
Lec. 3-A: Coding with AI

Zhigang Feng

Spring 2026

Introduction: The Research Architect Approach

The Core Message

You can implement sophisticated macro models with AI assistance, even if coding from scratch feels difficult. The key is mastering five foundational areas:

1. **Economic Theory**: Optimization problems, equilibrium concepts, optimality conditions
2. **Algorithm Design**: Value function iteration, Euler equation methods, policy iteration
3. **Numerical Techniques**: Optimization, interpolation, discretization, simulation
4. **Advanced Methods**: Perturbation, projection methods, and their trade-offs
5. **Code Literacy**: Reading code, understanding what it delivers, spotting issues

Once these foundations are solid, the actual coding can be **assisted or outsourced to AI** via tools like Cursor, Claude Code, and Claude Skills. You become the **research architect**—AI becomes your implementation partner.

What's Changing in the AI Era

AI tools can now write code from descriptions, debug errors, translate math into implementations, and refactor existing code. This shifts where you spend your time:

Traditional Approach

- Write every line yourself
- Debug syntax errors manually
- Hours on implementation details
- Bottleneck: *coding skill*

Research Architect

- Design algorithms precisely
- Direct AI implementation
- Validate economic correctness
- Bottleneck: *understanding*

Your comparative advantage lies in knowing *what question you want to answer*. AI can write code and check whether it is correct, but only you know what problem you're trying to solve and whether the answer makes economic sense.

The Five Pillars of Knowledge

Pillar 1: Economic Theory

You must be able to **formulate the economic problem** precisely:

Household optimization:

- Write the objective: $\max \mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t u(c_t, n_t)$
- Specify constraints: budget, borrowing limits, information structure
- Derive optimality conditions: Euler equations, transversality

Equilibrium concepts:

- Define competitive equilibrium: optimality + market clearing
- Understand recursive competitive equilibrium formulation
- Know when to use partial vs. general equilibrium

Why this matters: Without a precisely formulated problem, no amount of AI coding skill will produce useful results. You must specify the problem; AI helps solve it.

Pillar 2: Algorithm Design

You must know **how to translate theory into computation**:

Dynamic programming approach:

- Bellman equation formulation: $V(s) = \max_a \{u(s,a) + \beta \mathbb{E}[V(s')]\}$
- Value function iteration: iterate $T(V) \to V^*$ until convergence
- Policy function iteration: faster convergence, more complex per step

Euler equation methods:

- Time iteration on Euler equations directly
- Endogenous grid method (EGM): invert Euler equation for speed

Note: These are foundational examples to get you started. In practice, you'll learn many more methods (parameterized expectations, monotone operators, etc.). The key is understanding enough to choose appropriately and communicate effectively with AI.

Pillar 3: Numerical Techniques

You must understand **how computation works** and potential pitfalls:

Core techniques:

- **Optimization**: Grid search, Newton methods, choice of solver for FOCs
- **Interpolation**: Linear, cubic spline, Chebyshev—accuracy vs. speed trade-offs
- **Discretization**: Tauchen, Rouwenhorst for AR(1) processes; grid design
- **Simulation**: Monte Carlo for moments, ergodic distribution computation

Common issues you must recognize:

- Occasionally binding constraints (borrowing limits, irreversibility)
- Curse of dimensionality in high-dimensional state spaces
- Interpolation errors near kinks in policy functions

Why this matters: When AI-generated code produces wrong results, you diagnose whether it's a grid problem, interpolation error, or algorithm bug.

Pillar 4: Advanced Numerical Methods

For larger or more complex models, you need **advanced solution techniques**:

Perturbation methods:

- Linear approximation around steady state (1st order)
- Higher-order for welfare, risk premia (2nd, 3rd order)
- Fast but local—accuracy degrades far from steady state

Projection methods:

- Global approximation using basis functions (Chebyshev, finite elements)
- Handles nonlinearities and occasionally binding constraints
- Computationally intensive, requires careful convergence checks

Note: Again, these are illustrative. You should develop familiarity with multiple methods. Understanding trade-offs (speed vs. accuracy, local vs. global) lets you choose wisely and guide AI implementation.

Pillar 5: Code Literacy

You must be able to **read and understand code**, even if you don't write it from scratch:

What code literacy means:

- Read a VFI loop and verify it matches your algorithm design
- Identify where the Bellman equation is evaluated, how expectations are computed
- Spot potential issues: wrong indexing, missing constraints, numerical instability
- Understand data structures: arrays, grids, interpolation objects

Language familiarity (Python focus):

- NumPy arrays and vectorized operations
- Control flow: loops, conditionals, function definitions
- Common libraries: SciPy (optimization, interpolation), Numba (speed)

Why this matters: AI writes code; you verify it implements what you designed. Without code literacy, you cannot validate AI output or diagnose problems.

From Foundations to AI-Assisted Coding

You Provide (Foundations)	**AI Provides (Implementation)**
Research question and model	Syntactically correct code
Algorithm choice and design	Translation to Python/Julia/etc.
Numerical method selection	Optimization, interpolation routines
Convergence/accuracy criteria	Diagnostic outputs and plots
Economic interpretation of results	Debugging and refactoring

Once you master the five pillars, you can effectively direct AI to handle implementation. The division of labor is clear: **you are the architect**, AI is the builder. You design the structure and inspect the construction; AI handles the bricks and mortar.

AI Tools for Implementation

AI Tools: Overview

With foundations in place, you can leverage AI tools for implementation:

Type	**Examples**	**Best For**
Chatbox	Claude, ChatGPT	Algorithm design, theory questions, debugging with full context
IDE-Integrated	Cursor, Copilot	Real-time coding assistance, inline suggestions, file-aware edits
Autonomous Agent	Claude Code, Aider	Multi-file projects, large refactoring, autonomous iteration

Recommended workflow: Use Claude chatbox for algorithm design and deep debugging. Use Cursor for day-to-day implementation. Use Claude Code for larger autonomous tasks like converting legacy codebases.

Tool 1: Claude Chatbox (Design and Reasoning)

Claude's chatbox interface excels at algorithm design and deep reasoning:

Strengths:

- Large context window—can process papers, code, and detailed specifications together
- Strong mathematical and economic reasoning
- Can discuss trade-offs: "Should I use VFI or EGM for this problem?"

Best uses:

- **Algorithm design**: "Design a VFI algorithm for Aiyagari with these features..."
- **Paper implementation**: Upload PDF, ask "Explain and implement Section 3's algorithm"
- **Debugging**: Provide code + error + economic context for diagnosis

Note: AI capabilities evolve rapidly. Context window sizes, reasoning abilities, and tool integration improve frequently. The principle remains: use chatbox for design-level thinking.

Tool 2: Cursor (IDE Integration)

Cursor integrates AI directly into VS Code, providing real-time assistance as you implement:

Key features:

- **Inline completion** (Tab): AI suggests code as you type based on context
- **Chat** (Cmd/Ctrl+L): Ask questions with full file context—"Why is this loop slow?"
- **Composer** (Cmd/Ctrl+I): Describe changes in natural language, AI edits multiple files

Typical workflow:

1. Design your algorithm in Claude chatbox (get the logic right)
2. Open Cursor, describe the algorithm, let AI generate initial code
3. Use inline completion to fill in details; use Chat to debug
4. Run code, validate results, iterate

Best practice: Always provide economic context, not just code snippets.

Tool 3: Claude Code (Autonomous Agent)

Claude Code is a terminal-based agent that works autonomously on larger tasks:

What it does:

- Reads and writes multiple files across your project
- Runs code, observes errors, debugs and fixes autonomously
- Continues iterating until the task is complete (or asks for guidance)

Best uses:

- Converting a MATLAB codebase to Python
- Implementing a complete algorithm from a paper description
- Large-scale refactoring (e.g., restructuring a project)

Key difference from Cursor: Cursor assists your coding in real-time; Claude Code works autonomously while you supervise. You provide the specification and validation criteria; it handles the implementation loop.

The Iterative Development Workflow

The Iterative Workflow: Core Principle

Regardless of which AI tool you use, follow this systematic approach:

Design Algorithm $\rightarrow$ Implement Code $\rightarrow$ Validate $\rightarrow$ Extend $\rightarrow$ Repeat

Key distinction: Separate **algorithm design** from **code implementation**.

- First, design the algorithm on paper or in discussion with AI (state space, Bellman equation, solution method, convergence criterion)
- Then, implement the designed algorithm in code
- These are different activities requiring different thinking

Start simple: Begin with the simplest model version that has an analytical benchmark. Validate thoroughly. Add features one at a time, validating at each step. This catches bugs early and builds confidence incrementally.

The Five-Step Workflow

Step 1: Design Algorithm (before any coding)

- Specify state space, Bellman equation, solution method, convergence criterion
- **Check extensibility**: "Can this design accommodate the next feature?"

Step 2: Implement Code

- Translate algorithm to code; include diagnostics (convergence plots, Euler residuals)

Step 3: Validate Thoroughly

- Technical: convergence, no NaN, grid adequacy
- Economic: sensible results, correct comparative statics, limiting cases

Step 4: Extend Algorithm Design

- Add *one* feature; update algorithm design *before* coding

Step 5: Repeat

- Each validated version becomes the foundation for the next

Why This Workflow Works

1. Separating design from implementation catches conceptual errors early

Algorithm bugs are cheaper to fix on paper than in code. Design first, then implement.

2. Clear attribution of bugs

Adding one feature at a time means you know exactly what caused any new issue.

3. AI succeeds with structured, well-specified instructions

Good communication makes good collaboration. When you provide clear algorithm specifications with precise inputs, outputs, and success criteria, AI delivers better results. Vague requests produce vague code.

4. Analytical benchmarks catch errors early

The simplest version often has closed-form solutions. Validating against these catches fundamental errors before complexity obscures them.

Example: RBC Model

Example 1: RBC Model

Goal: Implement a stochastic RBC model with variable labor supply.

Don't start with the full model! Instead, build incrementally:

Version	Features	Benchmark
0	Deterministic, fixed labor	Closed-form steady state
1	Add stochastic productivity	Limiting case = Version 0
2	Add variable labor supply	Limiting case = Version 1

Version 0 specifics: Log utility, Cobb-Douglas production, VFI solution. Steady state is $k^* = \left(\frac{\alpha\beta}{1-\beta(1-\delta)}\right)^{\frac{1}{1-\alpha}}$. We use VFI even for the deterministic case because the algorithm structure extends naturally to the stochastic version.

RBC: Design Algorithm First

Step 1: Design algorithm (before any coding)

Your prompt to AI: "Design a VFI algorithm for deterministic RBC with: (1) Production $Y = K^\alpha$, depreciation δ; (2) Bellman equation $V(k) = \max_{k'} \log(K^\alpha + (1-\delta)K - k') + \beta V(k')$; (3) Grid search over k'."

Before implementing, check extensibility:

"Can this algorithm design extend to: (a) stochastic productivity with state (k, A)? (b) variable labor as additional choice? Show where these would enter the algorithm."

Why? If the algorithm structure can't accommodate shocks, you'll redesign later. Better to get the design right first. The algorithm should have clear places where you'd add the expectation operator and additional choice variables.

RBC: Implement and Validate

Step 2: Implement code with diagnostics

"Implement this algorithm in Python. Include: (1) Convergence plot: sup norm $|V^{n+1} - V^n|$ vs iteration; (2) Compare numerical k^* to analytical formula; (3) Plot policy $k'(k)$ with 45-degree line; (4) Euler equation residuals."

Step 3: Validate (your job, not AI's)

- Does $k'(k)$ cross the 45-degree line at analytical k^*?
- Are Euler residuals near machine precision ($< 10^{-8}$)?
- Do comparative statics work? (Higher $\beta \Rightarrow$ higher k^*)

Step 4-5: Extend and repeat

After Version 0 validates, design the stochastic extension (add AR(1) productivity, Tauchen discretization). Implement. Validate that setting $\sigma = 0$ recovers Version 0. Continue to variable labor.

Example: Aiyagari Model

Example 2: Aiyagari Model (Partial Equilibrium)

Goal: Solve the household problem in Aiyagari (1994) with idiosyncratic income risk.

Model: Households maximize $\mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t u(c_t)$ subject to $c + a' = (1+r)a + wz$ and $a' \geq \underline{a}$, where z is idiosyncratic productivity following a Markov process.

Incremental build:

Version	Features	Benchmark
0	Deterministic income ($z = 1$)	Three analytical cases
1	Stochastic income	Limiting case $=$ Version 0
2	Stationary distribution	Distribution sums to 1

Partial equilibrium: r and w taken as given. General equilibrium comes later.

Aiyagari: Version 0 and Extensions

Version 0: Deterministic savings

Three analytical benchmarks based on $\beta(1+r)$:

- $\beta(1+r) = 1$: Constant consumption; $a'(a) = a$
- $\beta(1+r) < 1$: Decumulate; $a \to \underline{a}$
- $\beta(1+r) > 1$: Accumulate; a grows

All three cases must match theory before proceeding.

Version 1: Add income uncertainty

Design algorithm with state (a, z), Tauchen discretization. Validate that $\sigma_z = 0$ recovers Version 0. Check precautionary savings: $\mathbb{E}[a]$ exceeds deterministic steady state.

Version 2: Stationary distribution

Compute $\mu(a, z)$ from policy function. Validate: sums to 1, mass at constraint, higher $\sigma_z \Rightarrow$ more inequality.

Validation Principles

Validation: Your Core Responsibility

AI writes code. **You validate economics.** At every iteration:

Technical Checks

- Convergence achieved?
- No NaN/Inf values?
- Policy not hitting grid boundaries?
- Euler residuals small?

Economic Checks

- Results economically sensible?
- Comparative statics correct?
- Limiting cases recover benchmarks?
- Matches analytical solutions?

Debugging with AI: Provide full context (code + error + expected behavior + economic intuition). AI suggests diagnostics; you run them and report back. AI catches indexing errors and numerical issues; you catch economic nonsense.

Claude Code: Encoding Your Workflow

The Evolution of Coding Assistance

Two decades ago (C++/Fortran):

- Write functions yourself: interpolation, optimization, rootfinding
- Debug low-level issues (integer division: $1/3 = 0$, not 0.333)

A few years ago (Python/Julia):

- Call library functions: `scipy.optimize`, `numpy.interp`
- Focus on algorithm logic, not implementation details

Recently (AI chatbox, Cursor):

- Describe the algorithm; AI writes the code that calls the libraries
- You validate results; AI debugs and iterates

Now (Claude Code with skills, agents, and rules):

- Encode your *entire methodology* once—AI follows it every time
- Specialized agents review, test, and fix autonomously
- Quality gates enforce standards before code ships

Two Platforms for Encoding Workflows

Claude offers two systems for persisting your workflow instructions:

	Claude.ai Projects	**Claude Code**
Interface	Web browser	Terminal (CLI)
Instructions	Project knowledge base	`CLAUDE.md` + `.claude/` folder
Invocation	Always loaded in project	Skills: `/command`; Rules: auto-loaded
Code execution	No	Yes (bash, scripts, compilers)
Multi-agent	No	Yes (parallel subagents)
Best for	Design, reasoning, analysis	Implementation, testing, workflows

We focus on **Claude Code**—the system that can run your code, spawn specialized reviewers, and iterate autonomously.

The Conceptual Map: Claude Code vs. Python

Every Claude Code component maps to a familiar programming concept:

Python	**Claude Code**	**Location**
Functions	**Skills** (slash commands)	`.claude/skills/`
Classes	**Agents** (specialized workers)	`.claude/agents/`
Subprocess / thread	**Subagents** (delegated tasks)	Spawned at runtime
Config files (`.yaml`)	**Rules** (auto-loaded standards)	`.claude/rules/`
`main()`	**Orchestrator**	Rule: `orchestrator-protocol.md`
Database / state	**Memory**	`CLAUDE.md`, `MEMORY.md`
Log files	**Session logs**	`quality_reports/`
`while` / `if`	**Natural language control flow**	Skill and agent definitions
`assert` / tests	**Quality gates** (80/90/95)	Rule: `orchestrator-protocol.md`
Decorators	**Hooks** (event triggers)	`settings.json`

Key insight: You are writing a "program" in natural language. The same design principles apply: modularity, separation of concerns, reusability.

The Workflow at a Glance

You $\rightarrow$ Instruction $\rightarrow$ Rules + Orchestrator $\rightarrow$ Skill Workflow $\rightarrow$ Agents/Subagents $\rightarrow$ Output

Two entry points:

- **Slash command** (`/solve-model ...`): directly selects that skill.
- **Plain English**: Claude plans under active rules, then executes directly or chooses a skill.

Direction of control: Rules $\rightarrow$ skill/workflow $\rightarrow$ agents. Agents do **not** choose skills.

Python analogy: Like `controller(input) -> workflow() -> workers() -> result`.

What a Project Looks Like

```
my-macro-project/
+-- CLAUDE.md                    # Project brain (read every session)
+-- MEMORY.md                    # Corrections: [LEARN:tag] entries
+-- .claude/
|   +-- settings.json            # Permissions for Python, git, etc.
|   +-- skills/                  # Workflows you invoke (/command)
|   |   +-- solve-model/         #   /solve-model aiyagari v0
|   |   |   +-- SKILL.md         #   Workflow: design -> implement -> validate
|   |   |   +-- references/      #   Algorithm specs and benchmarks
|   |   |       +-- rbc-guide.md
|   |   |       +-- aiyagari-guide.md
|   |   +-- validate/            #   /validate aiyagari v1
|   |       +-- SKILL.md
|   +-- agents/                  # Specialized reviewers
|   |   +-- code-reviewer.md     #   Code quality, vectorization
|   |   +-- domain-reviewer.md   #   Economic correctness
|   |   +-- verifier.md          #   Runs code, checks output
|   +-- rules/                   # Standards (always active)
|       +-- plan-first-workflow.md
|       +-- orchestrator-protocol.md
|       +-- iterative-workflow.md
|       +-- verification-protocol.md
|       +-- code-conventions.md
+-- scripts/                     # Python code (aiyagari_v0.py, ...)
```

Skills = Callable Workflows (The Orchestrator)

A **skill** is a multi-step workflow invoked by name. It is the component that ties everything together:

Python function:

- Defined once, called by name
- Takes arguments
- Executes a defined sequence
- Calls other functions / classes

Claude Code skill:

- Defined in `SKILL.md`
- Invoked: `/skillname arg`
- Follows step-by-step instructions
- References agent files by name

A skill's steps reference agents:

- Step 3: "Run the verifier agent" → Claude reads `verifier.md`, follows its checklist
- Step 4: "Run code-reviewer and domain-reviewer" → Claude reads both, runs them (possibly in parallel as subagents)

Control is one-way: skills call agents; agent files never choose which skill runs.

Skills are the glue. Rules constrain behavior; agents define roles; *skills* sequence them into a workflow

Agents = Role Descriptions (Not a Dispatch System)

An **agent file** is a markdown document that describes a specialized role—its checklist, review criteria, and report format. Claude Code does **not** have a native agent dispatcher. Agents run only when a skill (or an always-on rule like the orchestrator) references them. What actually happens:

1. A **skill/rule** says: "spawn the code-reviewer agent"
2. Claude **reads** `.claude/agents/code-reviewer.md`
3. Claude either:
   (a) **Adopts the role itself**—follows the checklist in the current conversation, or
   (b) **Launches a subagent** via the Task tool—a separate Claude instance that receives the agent file as its prompt

Why separate files? A single prompt checking vectorization, equilibrium conditions, and convergence does a mediocre job at everything. Separate agent files let Claude focus on one dimension at a time.

Subagent = child process. Option (b) runs in its own context window. Multiple subagents can run **in parallel**, like Python's `multiprocessing`. The result is returned to the main conversation.

Rules = Always-On Guardrails

Rules are the only component that requires zero user action:

- Every `.md` file in `.claude/rules/` is loaded at session start
- Claude obeys them throughout the entire session—you never invoke them
- They constrain *all* work: skill-invoked, plain English, or ad-hoc editing

Memory = persistent state across sessions:

- `CLAUDE.md`: project context, conventions, folder structure (read every session)
- `MEMORY.md`: corrections from past sessions (`[LEARN:tag]` format)
- Session logs: why decisions were made (not just what changed)

Hooks = event-triggered scripts (like decorators):

- Fire automatically on events (e.g., after every Claude response)
- Enforce behaviors that must survive long conversations

Key distinction: Rules tell Claude how to behave. Memory tells Claude what it knows. Hooks enforce behavior mechanically (shell scripts, not natural language).

Execution: Phase 1 — You Give an Instruction

Claude Code starts by reading `CLAUDE.md` — your project's "README for AI":

```
CLAUDE.md tells Claude:
  - What this project is (AI & ML for Quant Macro)
  - Available skills: /solve-model, /validate
  - Available agents: code-reviewer, domain-reviewer, verifier
  - Rules in .claude/rules/ (auto-loaded every session)
  - Folder structure, conventions, model versions (v0-v2)
```

Then you give an instruction — two styles:

```
Plain English:  "Implement the Aiyagari model with income risk."
Invoke a skill: /solve-model aiyagari
```

- **Plain English:** Claude plans under your rules and may execute directly or select an existing skill.
- **/solve-model:** Claude follows *that exact* encoded workflow (`SKILL.md`).
- **Agent selection:** happens downstream inside the active workflow; agents never select skills.
- **Either way:** Rules and orchestrator activate automatically

Python analogy: `CLAUDE.md` $\approx$ `__init__.py` + `pyproject.toml`. It bootstraps Claude's

Execution: Phase 2a — Rules Activate

Rules load automatically from `.claude/rules/` (NOT from SKILL.md):

```
.claude/rules/                  <- all .md files auto-loaded
  plan-first-workflow.md        -> "plan before coding"
  orchestrator-protocol.md      -> "implement-verify-review loop"
  iterative-workflow.md         -> "start simple, validate, extend"
  verification-protocol.md      -> "always run code before reporting"
  code-conventions.md           -> "set.seed(), no hardcoded paths"
```

The three layers, in activation order:

1. **Rules** = `.claude/rules/*.md`. Loaded automatically. Always active. You never invoke them. They constrain *how* Claude behaves.
2. **Skills** = `.claude/skills/*/SKILL.md`. Invoked by you with `/command`. They define *what* Claude does.
3. **Agents** = `.claude/agents/*.md`. Referenced by skills/rules. They define *who* does a sub-task (what role Claude adopts).

Python analogy: Rules ≈ config loaded at import time. Skills ≈ functions you call. Agents ≈ class

Execution: Phase 2b — Planning

The `plan-first-workflow.md` rule detects a non-trivial task:

```
Claude: "This is non-trivial. Let me plan first."

  Plan:
    Version 0: Deterministic savings (3 analytical benchmarks)
    Version 1: Add income risk (Tauchen discretization)
    Version 2: Stationary distribution

  Saves plan -> quality_reports/plans/2026-02-08_aiyagari.md

Claude: "Here is my plan. Approve?"

You: "Approved."
```

Why save the plan to a file? Claude's conversation window has limited memory. Saving the plan to disk ensures it survives even very long sessions.

Python analogy: Planning $\approx$ writing pseudocode before coding. The saved plan file $\approx$ a design document that persists beyond the current REPL session.

Execution: Phase 3 — Orchestrator Loop

The orchestrator is `.claude/rules/orchestrator-protocol.md` — a rule, not a command. It activates automatically after plan approval:

```
--- For each version (V0, V1, V2): ---

Step 1: IMPLEMENT  -> writes scripts/aiyagari_v0.py
Step 2: VERIFY     -> runs code, checks convergence
Step 3: REVIEW     -> spawns agents (code-reviewer,
                      domain-reviewer) in parallel
Step 4: FIX        -> applies agent suggestions
Step 5: SCORE      -> 85/100 >= 80 threshold? Done.

--- Repeats loop for Version 1, then Version 2 ---
```

Execution: Phase 3 — Orchestrator Loop (cont.)

Where does this logic live?

- The loop structure is in `.claude/rules/orchestrator-protocol.md` (a rule—always active)
- Agent roles are defined in `.claude/agents/*.md` (inert until referenced)
- The skill provides domain-specific steps that reference agents by name

What "spawns agents" actually means: The rule says "run code-reviewer." Claude reads `code-reviewer.md`, adopts its role (or launches a subagent), follows its checklist, and returns a report. The agent file is a prompt template, not a running process.

Execution: Phase 4 — Results and Learning

```
Claude: "All 3 versions complete. Score: 88/100.
  - V0: k* = 3.42 matches analytical formula
  - V1: precautionary savings confirmed
  - V2: distribution sums to 1.0
  Code: scripts/aiyagari_v0.py, v1.py, v2.py"

You: "The comparative statics for beta look wrong."

Claude: [investigates -> grid too coarse near constraint]
        [fixes grid, re-runs, validates]
        [saves to MEMORY.md:]
          [LEARN:aiyagari] Grid needs finer spacing
          near borrowing constraint for accurate
          comparative statics.
```

Next session: Claude reads MEMORY.md and never makes the same mistake again.

Python analogy: MEMORY.md $\approx$ a persistent database of lessons learned. Each [LEARN:tag] entry is like adding a regression test that prevents future regressions.

Your Role: What You Build

You are the architect. Claude Code is powerful, but it needs your project structure to follow your methodology. Two options:

Option 1: Start from a template

- Fork an existing workflow project (like the one in this course)
- Modify skills, agents, and rules for your domain
- Add your model-specific reference files

Option 2: Build from scratch

- Create the `.claude/` folder structure
- Write skills for your common workflows
- Write agents for your review dimensions
- Write rules for your standards

The key investment: Writing good skills and rules takes a few hours upfront, but saves you from re-explaining your methodology in every future session. Like writing a library: effort once, reuse forever.

Skill File Structure

A skill lives in `.claude/skills/[name]/` with a required `SKILL.md`:

```
.claude/skills/solve-model/
  ⊢ SKILL.md              (required:  workflow steps)
  ⌞ references/            (optional:  detailed guides)
      ⊢ rbc-guide.md     (RBC algorithm details)
      ⌞ aiyagari-guide.md   (Aiyagari details)
```

SKILL.md has two parts:

1. **YAML frontmatter**: Name and description (determines when skill triggers)
2. **Markdown body**: Workflow instructions, validation criteria, prompting patterns

Reference files contain detailed model-specific algorithms. Claude loads them only when needed, keeping context efficient.

Agent File Structure

An agent is a **single** `.md` **file** in `.claude/agents/`—a role description, not executable code:

```
.claude/agents/
  ⊢ code-reviewer.md    (vectorization, stability, style)
  ⊢ domain-reviewer.md  (equations, equilibrium, intuition)
  ∟ verifier.md          (runs code, checks convergence)
```

Each file specifies:

- **What** it reviews (code quality? domain correctness? convergence?)
- **How** it reports (severity levels, structured format)
- **Permissions** (reviewers are read-only; verifier can execute code)

These files are inert until referenced. They sit in the folder doing nothing. When a skill step says "spawn the code-reviewer," Claude reads the file, adopts its role, and follows its checklist.

Python analogy: Agent file ≈ class definition. Not instantiated until called. Multiple agents can be

Rule File Structure

A rule is a **single** `.md` **file** in `.claude/rules/`:

```
.claude/rules/
  ⊢ plan-first-workflow.md    (plan before coding)
  ⊢ orchestrator-protocol.md   (implement-verify-review loop)
  ⊢ iterative-workflow.md    (start simple, extend)
  ⊢ verification-protocol.md   (always test before reporting)
  ∟ code-conventions.md     (set.seed, no hardcoded paths)
```

Rules are **always active**. You never invoke them. Claude reads every rule file at the start of every session and obeys them throughout.

Python analogy: Rules ≈ `pyproject.toml` + linter configs. They constrain all code without being called explicitly.

How Skills, Agents, and Rules Connect

	Rules	**Skills**	**Agents**
Location	`.claude/rules/`	`.claude/skills/`	`.claude/agents/`
Loaded	Automatically, always	When you type `/cmd`	When a skill/rule references them
Purpose	*How* to behave	*What* to do	*Who* does a sub-task
Mechanism	Claude obeys	Claude follows steps	Claude reads & adopts role
Python $\approx$	Config / linter	Function	Class definition
Example	"plan before coding"	`/solve-model`	`code-reviewer.md`

The causal chain:

1. **Rules** are loaded at session start—they constrain everything that follows
2. You invoke a **skill** (or give plain English and Claude plans a workflow)
3. The active workflow says "use the code-reviewer agent"—Claude reads the **agent file** and follows its checklist (or launches a subagent with those instructions)

Rules are always on. Skills are user-invoked or workflow-selected. Agents are downstream workers referenced by skills/rules.

Tracing the Flow: `/solve-model aiyagari` (1/2)

```
my-macro-project/
+-- CLAUDE.md                    # 1. Read first: project context
+-- MEMORY.md                    # 2. Read: past corrections
+-- .claude/
|   +-- rules/                   # 3. ALL 5 rule files auto-loaded
|   |   +-- plan-first-workflow.md
|   |   +-- orchestrator-protocol.md
|   |   +-- iterative-workflow.md
|   |   +-- verification-protocol.md
|   |   +-- code-conventions.md
|   +-- skills/
|   |   +-- solve-model/
|   |       +-- SKILL.md         # 4. Read: you typed /solve-model
|   |       +-- references/
|   |           +-- aiyagari-guide.md  # 5. Read: SKILL.md says "see ref"
|   +-- agents/
|       +-- code-reviewer.md     # 6. Spawned later by orchestrator
|       +-- domain-reviewer.md   # 6. Spawned later by orchestrator
|       +-- verifier.md          # 6. Runs scripts, checks output
+-- scripts/                     # 7. Code written here
+-- output/                      # 7. Results saved here
+-- quality_reports/plans/        # 8. Plans saved here
```

Numbers show the **reading order**. Claude reads 1–5 before writing any code; 6–8 happen during the

Tracing the Flow: `/solve-model aiyagari` (2/2)

```
You type: /solve-model aiyagari

Step 1: Read CLAUDE.md              -> knows project, conventions
Step 2: Read MEMORY.md              -> recalls past corrections
Step 3: Load .claude/rules/*        -> all constraints active
Step 4: Read solve-model/SKILL.md  -> gets your workflow steps
Step 5: SKILL.md says "see references/aiyagari-guide.md"
        -> reads detailed algorithm specs

--- plan-first-workflow.md rule kicks in ---
Step 6: Write plan, save to quality_reports/plans/
Step 7: Ask for approval -> You: "Approved."

--- orchestrator-protocol.md rule activates ---
Step 8:  IMPLEMENT  -> writes scripts/aiyagari_v0.py
Step 9:  VERIFY     -> spawns verifier.md agent
                       (runs script, checks convergence, no NaN)
Step 10: REVIEW     -> spawns code-reviewer.md agent
                       spawns domain-reviewer.md agent
Step 11: FIX        -> applies their suggestions
Step 12: SCORE      -> Correctness >= 8? Done. Next version.
```

Every step maps to a specific file in the project folder.

Inside CLAUDE.md

`CLAUDE.md` is Claude's "README for AI" — read at the start of every session:

```
# CLAUDE.md --- Aiyagari Model Project

**Project:** AI & ML for Quant Macro
**Language:** Python (NumPy/SciPy)
**Philosophy:** Never skip validation between versions.

## Available Skills
| Command                | What It Does                     |
| /solve-model aiyagari  | Iterative model-building workflow |
| /validate aiyagari v1  | Run benchmarks and convergence   |

## Available Agents
| Agent           | Mode      | Purpose                    |
| code-reviewer   | READ-ONLY | Vectorization, stability   |
| domain-reviewer | READ-ONLY | Equations, equilibrium     |
| verifier        | EXECUTE   | Runs scripts, checks output |

## Model Versions
v0: Deterministic savings  |  v1: Income risk  |  v2: GE
```

Inside MEMORY.md

`MEMORY.md` stores corrections so Claude never repeats the same mistake:

```
# MEMORY.md --- Accumulated Learnings

[LEARN:aiyagari-grid] Asset grid should use more points
  near the borrowing constraint. Triple-exponential
  spacing with 200 points. Linear grids need 500+.

[LEARN:vfi-howard] Howard improvement cuts VFI iterations
  by ~80%. After every 25 VFI iterations, run 50 Howard
  steps. Convergence tolerance: 1e-8 on sup-norm.

[LEARN:notation] Convention: V for value function,
  c_pol for consumption policy, a_pol for savings policy,
  Pi for transition matrix, mu for stationary distribution.
```

How entries are created: When you correct Claude ("the grid is too coarse"), Claude saves a `[LEARN:tag]` entry. Next session, it reads these before starting work.

Python analogy: `MEMORY.md` $\approx$ a persistent database of regression tests. Each entry prevents a known failure from recurring.

SKILL.md: Frontmatter and Body

```
---
name: Solve Model
description: Iterative model building -- design, implement,
  validate, extend
command: solve-model
arguments:
  - name: model    # e.g., aiyagari, rbc
  - name: version  # v0, v1, v2
---
# Solve Model Skill

## Steps
### 1. Design Algorithm
Read references/{model}-guide.md. Save plan to quality_reports/.

### 2. Implement with Diagnostics
Write scripts/{model}_{version}.py. Include convergence,
benchmarks, figures. Follow rules/code-conventions.md.

### 3. Validate
Run verifier agent. Run domain-reviewer. Run code-reviewer.
ALL must pass before proceeding.

### 4. Extend
```

Inside a Reference File: `rbc-guide.md`

Reference files live inside a skill folder. They contain the detailed algorithm specs that `SKILL.md` links to:

```
# RBC Model -- Algorithm Reference

## Model: u(c) = c^(1-sigma)/(1-sigma), Y = z*K^alpha*L^(1-alpha)
## Params: sigma=2, alpha=0.36, beta=0.99, delta=0.025

## Version 0: Deterministic Steady State
- r_ss = 1/beta - 1,  K_ss = (alpha/(r_ss+delta))^(1/(1-alpha))
- Benchmark: K_ss = 11.812, C_ss = 0.8615

## Version 1: Stochastic TFP
- log(z') = rho*log(z) + eps, rho=0.95, sigma_z=0.007
- Tauchen: N_z=7, m=3 std devs. Grid: N_k=200
- Benchmark: E[K] ~ 11.81, std(Y)/std(z) ~ 1.4

## Version 2: Variable Labor
- FOC: chi*l^(1/eta) = w*c^(-sigma), eta=1.0
- Benchmark: E[L] ~ 1.0, labor procyclical
```

Claude loads reference files **only when needed**. Ask about RBC → reads rbc-guide.md. Ask about Aiyagari → reads aiyagari-guide.md. Keeps context efficient.

Inside an Agent File: `code-reviewer.md`

Agent files live in `.claude/agents/`. Each defines one specialist:

```
---
name: Code Reviewer
mode: read-only
tools: [Read, Glob, Grep]
---
# Code Reviewer Agent

You are a code quality reviewer for quantitative macro.
**You are READ-ONLY. Do not modify any files.**

## Review Checklist
- Numerical stability (no division by zero, convergence check)
- Vectorization (NumPy broadcasting, not Python for loops)
- Reproducibility (np.random.seed(42), no hardcoded paths)
- Clarity (key equations commented, conventions followed)

## Report Format
[SEVERITY] Description
  File: scripts/filename.py, Line: N
Severities: CRITICAL, HIGH, MEDIUM, LOW
```

Python analogy: An agent file ≈ a class definition. The orchestrator creates an instance when it

Inside a Rule File: `orchestrator-protocol.md`

Rule files live in `.claude/rules/`. They are always active — never invoked:

```
---
trigger: after any code implementation
priority: critical
---
# Orchestrator Protocol

Every code change follows: Implement -> Verify -> Review -> Fix

1. IMPLEMENT --- Write/modify code in scripts/
2. VERIFY    --- Run via verifier agent; check convergence, no NaN
3. REVIEW    --- Spawn code-reviewer AND domain-reviewer (parallel)
4. FIX       --- Apply fixes (Critical > High > Medium)
5. SCORE     --- Correctness >= 8, Code Quality >= 6

Max 3 fix iterations per step. Each loop must show improvement.

## Agent Selection
| Step    | Agent           | Mode     |
| Verify  | verifier        | EXECUTE  |
| Review  | code-reviewer   | READ-ONLY|
| Review  | domain-reviewer | READ-ONLY|
```

Using Skills: Three Approaches

Approach 1: Claude Code (most powerful)

- Place skill in `.claude/skills/[name]/SKILL.md`
- Invoke: `/skillname` in the terminal
- Full capabilities: run code, spawn agents, compile, test, iterate

Approach 2: Claude.ai Projects (web interface)

- Create a Project; add workflow files to its knowledge base
- Instructions loaded in every conversation within that Project
- Good for design and reasoning, but cannot execute code

Approach 3: Paste into conversation

- Copy instructions directly into a chat message
- Does not persist—must re-paste each session
- Good for one-off use or testing before formalizing

The Complete Architecture

Python	Claude Code	Activation	Role
Config / linter	Rules	Auto (session start)	Standards
Functions	Skills (`/cmd`)	You invoke	Workflow steps
Class definitions	Agents (`.md`)	Skills reference	Role templates
`multiprocessing`	Subagents	Skills launch	Parallel execution
`main()`	Orchestrator rule	Auto (always on)	Loop coordination
State / DB	`CLAUDE.md`	Auto (session start)	Project context
Logging	Session logs	During execution	Decision history
`while`/`if`	Natural language	In skill/rule text	Control flow
`assert` / tests	Quality gates	In orchestrator	Acceptance criteria
Decorators	Hooks	Event-triggered	Shell enforcement

The activation chain: Rules + Memory load automatically → you invoke a skill → the skill references agents → agents execute (or are launched as subagents) → quality gates in the orchestrator rule decide whether to iterate or stop.

Summary

The Complete Picture

Master the five pillars:

1. Economic theory: formulate problems, define equilibria
2. Algorithm design: VFI, EGM, policy iteration—and many more
3. Numerical techniques: optimization, interpolation, their pitfalls
4. Advanced methods: perturbation, projection, trade-offs
5. Code literacy: read code, verify it matches your design

Then leverage AI tools:

- Claude chatbox for algorithm design and debugging
- Cursor for day-to-day implementation
- Claude Code for autonomous larger tasks
- Claude Skills to encode your methodology for consistent reuse

Always follow the workflow: Design algorithm $\rightarrow$ Implement $\rightarrow$ Validate $\rightarrow$ Extend $\rightarrow$ Repeat.

Key Takeaway

You don't need to be an expert coder.

You need to understand theory, algorithms, and numerics well enough to **specify problems clearly**, **direct AI effectively**, and **validate results rigorously**.

AI democratizes implementation. Your unique value is knowing *what* to compute, *why* it matters economically, and *whether* the results are correct. Master the foundations, communicate clearly with AI, and it handles the rest.

Exercise: AI-Assisted Dynamic Equilibrium Modeling

Exercise Overview

Goal: Build a reusable framework for solving and extending dynamic equilibrium models using AI tools.

- **Starting point**: Lab11A notebook — Krusell-Smith (1998) solved with Deep Equilibrium Nets (DEQN) in PyTorch.
- **Workflow** (5 parts):
  1. **Document the model** — formal math in LaTeX
  2. **Document the algorithm** — pseudo-code for DEQN
  3. **Structure the code** — refactor notebook into modules
  4. **Create a Claude Code project** — skills, agents, rules
  5. **Iterate with extensions** — add features, validate, repeat
- This is a **semester project**, not a homework. AI tools (Claude Code, Cursor) are expected to help you accomplish it.

The Krusell-Smith (1998) Model — Overview

- **Heterogeneous agents** with incomplete markets and aggregate uncertainty.
- Why it is a benchmark:
  - Simplest model combining **idiosyncratic risk** (employment shocks) with **aggregate risk** (TFP shocks).
  - Requires tracking the **wealth distribution** as a state variable.
  - Classic test case for computational methods in macro.
- The Lab11A notebook solves this model using **Deep Equilibrium Nets** (DEQN) in PyTorch.

Your task
Understand this model deeply enough to **document it formally** and then **extend it**.

The Model — Households

Households maximize lifetime utility:

$$\max\ \mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t\, u(c_t), \quad u(c) = \frac{c^{1-\gamma}}{1-\gamma}$$

subject to:

- **Budget constraint**: $c_t + a_{t+1} = (1 + r_t)\, a_t + w_t\, e_t$
- **Borrowing constraint**: $a_{t+1} \geq \underline{a}$
- **Idiosyncratic productivity**: e_t follows a Markov chain with transition matrix Π_e

In the notebook: $\gamma = 1/\texttt{eis} = 2$, $\beta = 0.98$, $\underline{a} = 0$, and e takes 3 values (Rouwenhorst discretization).

The Model — Firms and Equilibrium

Representative firm with Cobb-Douglas production:

$$Y_t = Z_t\, K_t^{\alpha}\, L^{1-\alpha}$$

Competitive prices:

$$r_t = \alpha\, Z_t \left(\frac{K_t}{L}\right)^{\alpha-1} - \delta, \quad w_t = (1-\alpha)\, Z_t \left(\frac{K_t}{L}\right)^{\alpha}$$

Aggregate TFP shock: Z_t follows a Markov chain (Tauchen, 5 states, $\rho = 0.9$, $\sigma = 0.007$).

Recursive Competitive Equilibrium: A policy function $a'(a, e, Z, \mu)$, a price system (r, w), and a law of motion for μ (the wealth distribution) such that:

1. Households optimize given prices.
2. Firms maximize profits.
3. Markets clear: $K_{t+1} = \int a'(a, e, Z_t, \mu_t)\, d\mu_t$.

Part 1 — Document the Model

Instructions: Write a standalone LaTeX file that formally describes the model.

- **Include**:
  - Model primitives (preferences, technology, shocks)
  - Household problem (Bellman equation or sequential formulation)
  - Firm problem and competitive prices
  - Equilibrium definition (markets, consistency)
- **Tip**: Use AI to translate from the notebook's code comments to formal math. Paste the notebook cells into Cursor and ask: *"Write the formal economic model that this code implements."*
- **Deliverable**: `model_documentation.tex`

Why document first?
Writing the math forces you to understand the model **before** touching the code. This prevents "coding by copy-paste" and makes extensions much easier.

The DEQN Algorithm — Overview

Deep Equilibrium Nets (Azinovic, Gaegauf, Scheidegger, 2022):

- **Key idea**: A neural network approximates the policy function $\sigma(a, e, Z, K) \to$ savings rate.
- **Training**: Minimize Euler equation errors across simulated economies.
- **Three components**:
  1. **Initialization** — use Sequence-Space Jacobian (SSJ) to compute steady state and calibrate grids.
  2. **Policy network** — neural net maps individual + aggregate state to savings decisions.
  3. **Cloud simulation** — N independent copies of the economy evolve in parallel. Each copy n traces out a *sequence*:

$$(\mu_{n,0},\, Z_{n,0}) \;\to\; (\mu_{n,1},\, Z_{n,1}) \;\to\; \cdots \;\to\; (\mu_{n,T},\, Z_{n,T})$$

     advancing one period per training epoch. Aggregate capital $K_{n,t} = \int a \, d\mu_{n,t}$ is computed from the distribution, giving a batch of aggregate states $\{(K_n, Z_n)\}_{n=1}^{N}$ for training.

Why DEQN?
It avoids the "curse of dimensionality" that plagues grid-based methods. Adding state variables (new shocks, assets) requires only changing the NN input dimension — no exponential grid explosion.

The DEQN Algorithm — Initialization and Forward Pass

1. **Calibrate & steady state**: Use SSJ to solve for K_{ss}, grids, and the stationary distribution μ_{ss}.
2. **Initialize cloud**: N copies of μ_{ss}, each paired with a random TFP draw Z_n. All copies start identical; over training, they diverge as each experiences a different shock history.
3. **Forward pass**: Neural net maps $(a, e, Z, K) \to$ savings rate σ. Compute savings:

$$a' = \underline{a} + \sigma \cdot (\text{coh} - \underline{a})$$

4. **Euler errors**: Sample random agents, compute:

$$\varepsilon = 1 - \frac{\beta\, \mathbb{E}\left[(1 + r')\, u'(c')\right]}{u'(c)}$$

Steps 1–2 run once. Steps 3–4 run every training epoch.

The DEQN Algorithm — Training Loop

5. **Backpropagate**: Loss $= \mathbb{E}[\varepsilon^2]$. Update NN weights via Adam optimizer.
6. **Transport distribution**: Move $\mu_t \to \mu_{t+1}$ using the policy function (linear interpolation on asset grid + Markov transition for e).
7. **Repeat**: Update aggregate capital $K_{t+1} = \int a \, d\mu_{t+1}$, draw next Z, go to step 3.

The loop (steps 3–7) runs for 2000 epochs. The loss should decrease steadily; policy functions should stabilize.

Key insight
The neural network learns by minimizing Euler equation residuals — the same optimality condition you derived on paper. The cloud advances one period per epoch: after T epochs, each copy has traversed T time periods, generating sequences of distributions $\{\mu_{n,t}\}$ that explore a wide range of aggregate states (K, Z).

The Cloud — Why It Is Needed and How It Connects to Actor-Critic

Why a cloud? The wealth distribution μ is a state variable in the KS model (see Lec. 6, slide "Mapping Macro to RL": State $= (k, z, \Delta)$). To train the neural network, we need diverse aggregate states (K, Z), but $K = \int a \, d\mu$ depends on μ.

- **At epoch 0**: All N copies start at μ_{ss}, so $K_n \approx K_{ss}$. Only Z_n varies.
- **As training progresses**: Each copy's $\mu_{n,t}$ evolves under its own TFP shock sequence, causing the aggregate capitals $\{K_n\}$ to spread out across a realistic range.
- **Result**: The cloud generates a batch of diverse, economically realistic (K_n, Z_n) pairs — analogous to a training batch in deep learning.

Connection to Actor-Critic (Lec. 6)
Both DEQN and actor-critic use *simulation* to generate training data. In actor-critic, the T_{sim}-step return simulates a representative agent forward to reduce bias in value estimates (two networks: actor + critic). In DEQN, the cloud simulates N heterogeneous economies forward, using Euler equation residuals as the loss (one network, no critic needed). The cloud solves the intractable distribution transition $\mu' = H(\mu)$ identified in Lec. 6 by simply *simulating* it forward.

Part 2 — Document the Algorithm

Instructions: Add an algorithm section to your `model_documentation.tex`.

- Write each step in **pseudo-code** (not Python — language-agnostic).
- Document:
  - **Inputs**: calibration parameters, grid sizes, NN architecture.
  - **Outputs**: trained policy network, simulated distribution.
  - **Convergence criterion**: loss $< \epsilon$ or max epochs.
- **Map each step to the notebook cell** that implements it:
  - Step 1 (SSJ) $\rightarrow$ Cells 4–5
  - Step 2 (Network) $\rightarrow$ Cell 7
  - Step 3 (Physics) $\rightarrow$ Cell 8
  - Steps 4–7 (Training loop) $\rightarrow$ Cell 10
- **Deliverable**: Updated `model_documentation.tex` with algorithm section.

Part 3 — Structure the Code

The notebook is **monolithic** — all logic in one file. Restructure into modules:

File	**Responsibility**
`config.json`	All parameters: β, α, δ, grid sizes, NN architecture, training hyperparameters
`model.py`	`KrusellSmithModel` class: prices, utility, distribution transport
`network.py`	`KS_Network` class: neural net architecture
`train.py`	Training loop: cloud simulation, Euler errors, optimization
`dashboard.py`	Live monitoring: loss plots, policy function visualization, K dynamics
`main.py`	Entry point: load config, initialize, train, save results

Tip: Paste notebook cells into Cursor and ask: *"Refactor this into a class in model.py with methods for get_prices, utility_prime, and transport_distribution."*

Code Structure — `config.json` Example

```json
{
  "model": {
    "beta": 0.98,  "gamma": 2.0,
    "alpha": 0.36, "delta": 0.025,
    "rho_e": 0.966, "sd_e": 0.5,
    "rho_tfp": 0.9, "sigma_tfp": 0.007
  },
  "grid": {
    "nA": 100, "nE": 3, "nTFP": 5,
    "amin": 0, "amax": 50
  },
  "network": {
    "n_hidden": 64, "n_layers": 3,
    "activation": "mish"
  },
  "training": {
    "n_epochs": 2000, "batch_size": 256,
    "n_agents_sample": 64,
    "lr": 1e-3, "scheduler_step": 500,
    "scheduler_gamma": 0.5
  }
}
```

Code Structure — Key Design Principles

- **Separation of concerns**:
  - Economics logic (`model.py`) is separate from ML logic (`network.py`) and orchestration (`train.py`).
  - You can change the NN architecture without touching the economics, and vice versa.
- **Configuration-driven**:
  - Change parameters in `config.json` without modifying code.
  - Run experiments by swapping config files.
- **Dashboard for monitoring**:
  - Watch the loss curve, policy functions, and K dynamics evolve during training.
  - Catch problems early (divergence, flat loss, unreasonable policies).
- **Testability**: Each module can be tested independently. Does `model.get_prices(K, Z)` return the right values? Does the NN output have the right shape?

Part 4 — Create a Claude Code Project

Goal: Set up a `.claude/` directory so Claude Code can help you extend models systematically.

Using the Claude Code components introduced above, create:

- `CLAUDE.md` — project overview, folder structure, workflow rules, current state
- **Skills** (`.claude/skills/`) — slash commands for common operations
- **Agents** (`.claude/agents/`) — specialized reviewers
- **Rules** (`.claude/rules/`) — workflow constraints that are always enforced

Why this matters
Without a `CLAUDE.md`, the AI has no memory of your project structure. With one, it knows *exactly* where your model is, what your code does, and how to extend it safely.

Claude Code Project — Skills to Create

Skill	What it does
`/extend-model`	Add a feature to the model (new shock, preference, constraint). Updates `model_documentation.tex`.
`/extend-algorithm`	Update the DEQN algorithm for the extended model. Adjusts pseudo-code and identifies code changes needed.
`/extend-code`	Modify the Python codebase to implement the extension. Updates `model.py`, `network.py`, etc.
`/run-experiment`	Run training with a specific `config.json`. Save results and logs.
`/validate-model`	Check Euler errors, inspect policy function shapes, verify steady-state convergence.
`/document-extension`	Update `model_documentation.tex` after a code change is validated.

Each skill is a markdown file in `.claude/skills/` that tells Claude Code exactly what steps to follow.

Claude Code Project — Agents and Rules

Agents (specialized reviewers in `.claude/agents/`):

`model-reviewer`	Check economic logic: are the FOCs correct? Is the equilibrium definition consistent?
`code-reviewer`	Check Python/PyTorch: shapes, device placement, gradient flow, numerical stability.
`algorithm-reviewer`	Check numerical methods: convergence, interpolation accuracy, distribution transport.

Rules (always-on constraints in `.claude/rules/`):

`iterative-workflow.md`	Always extend minimally: one feature at a time.
`validation-protocol.md`	Always validate (Euler errors $< 10^{-3}$) before extending further.
`documentation-first.md`	Update `model_documentation.tex` before writing code.

Part 5 — The Iterative Extension Workflow

Start: Base KS model — working and validated.

Extension 1: Add labor supply choice (minimal change)

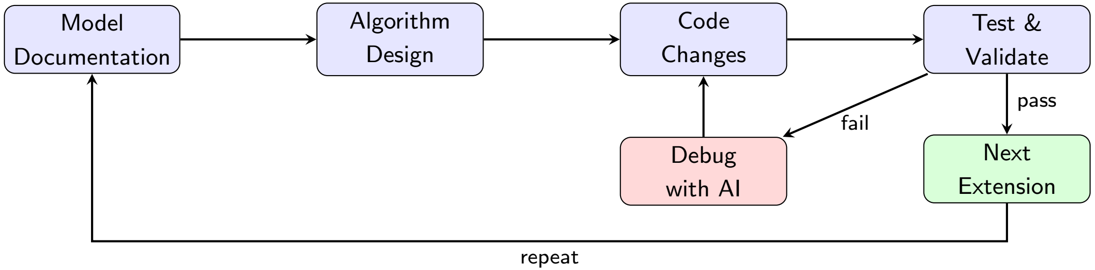
- **Model**: Add labor-leisure tradeoff to household problem.

$$u(c,\ell) = \frac{c^{1-\gamma}}{1-\gamma} - \psi \frac{\ell^{1+\phi}}{1+\phi}$$

- **Algorithm**: Neural net now outputs **two** decisions: (savings rate, labor supply).
- **Code**: Update `network.py` output dimension, add labor FOC to `model.py`.
- **Validate**: Does it converge? Are policy functions reasonable?

Extension 2: Add a second asset (bonds + capital).
Extension 3: Your research model.

The Iteration Cycle



This is the **“Research Architect”** workflow. Each cycle adds one feature, validates it, then moves on.

Deliverables

1. `model_documentation.tex` — formal model description + algorithm pseudo-code.
2. **Structured Python codebase** — 6 files:
   - `config.json`, `model.py`, `network.py`,
   - `train.py`, `dashboard.py`, `main.py`
3. **Claude Code project** — `.claude/` directory with:
   - `CLAUDE.md` (project guide),
   - at least 3 skills,
   - at least 2 agents,
   - at least 2 rules.
4. **At least one model extension** — documented, coded, and validated.
5. **Brief report** (1–2 pages): What worked? What didn't? How did AI help?

Tips for Working with AI

- **Start small**: Get the base model working and validated first. Do not jump to extensions until you understand the base.
- **Be specific**: Instead of *"refactor this code,"* say *"Refactor this cell into a class with methods `get_prices`, `utility_prime`, and `transport_distribution`."*
- **Validate everything**: AI-generated code can have subtle bugs in economic logic. Always check:
  - Do Euler errors converge below 10^{-3}?
  - Are policy functions monotone in wealth?
  - Does aggregate K stay near the steady state?
- **Document as you go**: The `.tex` file IS your thinking. If you cannot write down the math, you do not understand the model well enough to code it.
- **Use the iterative cycle**: Never jump ahead. One extension at a time.