

Reasoning LLMs and Coding Agents

DS542 — Deep Learning for Data Science

Based on two essays by Sebastian Raschka:

- [Understanding Reasoning LLMs \(Feb 2025\)](#)
- [Components of a Coding Agent \(Apr 2026\)](#)

Today's Roadmap

Part 1 — Reasoning LLMs

1. What is a "reasoning model"?
2. When should we (and shouldn't we) use one?
3. The DeepSeek-R1 family as a case study
4. Four ways to build/improve reasoning models
5. Doing it on a budget: Sky-T1, TinyZero, Journey Learning

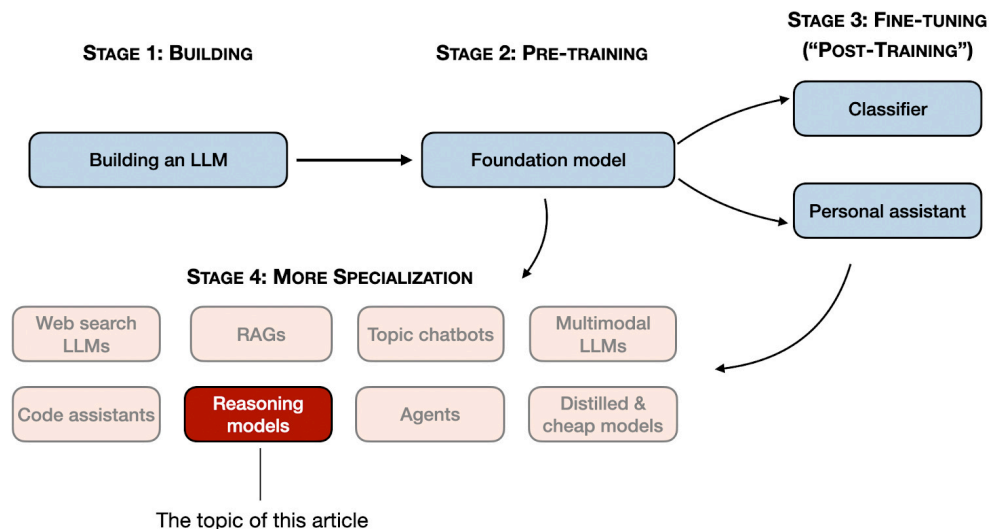
Part 2 — Coding Agents

1. LLM vs. reasoning model vs. agent
2. Six components of a coding harness
3. Why a good harness can matter more than the model

Wrap-up — takeaways, discussion

Part 1: Reasoning LLMs

Stage 4 of the LLM Lifecycle



1. **Pre-training** — next-token prediction on web-scale text
2. **Supervised fine-tuning (SFT)** — instruction following
3. **Preference tuning** — RLHF / DPO / etc.
4. **Specialization** — RAG, code assistants, **reasoning models**, ...

Reasoning models are one of the most important specializations to emerge in 2024–2025. Specialization *adds* capability — it does not replace general-purpose LLMs.

Defining "Reasoning"

No universally agreed-upon definition, but a working one:

Reasoning = answering questions that require complex, multi-step generation with intermediate steps.

Query

"What is the capital of France?"

"A train at 60 mph travels 3 hours. How far?"

"Prove that $\sqrt{2}$ is irrational."

Reasoning needed?

No — factual lookup

Yes — relate distance, speed, time

Yes — multi-step proof

Defining "Reasoning"

No universally agreed-upon definition, but a working one:

Reasoning = answering questions that require complex, multi-step generation with intermediate steps.

Query	Reasoning needed?
"What is the capital of France?"	No — factual lookup
"A train at 60 mph travels 3 hours. How far?"	Yes — relate distance, speed, time
"Prove that $\sqrt{2}$ is irrational."	Yes — multi-step proof

Also called "*thinking fast*" and "*thinking slow*".

Where Does "Reasoning" Show Up?

If a train is moving at 60 mph and travels for 3 hours, how far does it go?

The train travels 180 miles.

Plain response

To determine the distance traveled, use the formula:

Distance = Speed × Time

Given that the speed is 60 mph and the time is 3 hours:

Distance = 60 mph × 3 hours = 180 miles

So, the train travels 180 miles.

Response with intermediate reasoning steps

Intermediate steps can appear in two places:

1. **In the visible response** — the model writes out its work
2. **In hidden iterations** — e.g. OpenAI's o1 runs multiple internal passes; only the final answer is shown to the user

Most modern reasoning models do **both**. Non-reasoning LLMs can *also* produce intermediate steps when prompted — the difference is *default behavior* and *training*.

When Is a Reasoning Model the Right Tool?

Good fit

- Math proofs, competition problems
- Logic puzzles, riddles
- Complex / multi-file coding tasks
- Multi-step planning under constraints

Poor fit

- Summarization
- Translation
- Fact retrieval / basic Q&A
- Short-form creative writing

Reasoning models are **more expensive**, **more verbose**, and can **overthink** simple prompts.

Rule of thumb: use the right tool for the task.

Strengths & Weaknesses

Good at

- + Deductive or inductive reasoning (e.g., riddles, math proofs)
- + Chain-of-thought reasoning (breaking down multi-step problems)
- + Complex decision-making tasks
- + Better generalization to novel problems

Bad at

- Fast and cheap responses (more inference time)
- Knowledge-based tasks (hallucination)
- Simple tasks ("overthinking")

Strengths

Excellent on complex, multi-step problems
Self-verification / error correction
Better math & coding benchmarks
Produces inspectable reasoning traces

Weaknesses

Higher per-token cost
Longer responses, higher latency
Can "overthink" easy prompts
Inference costs scale with thought length

This trade-off is why providers offer *both* a regular and a "thinking" variant.

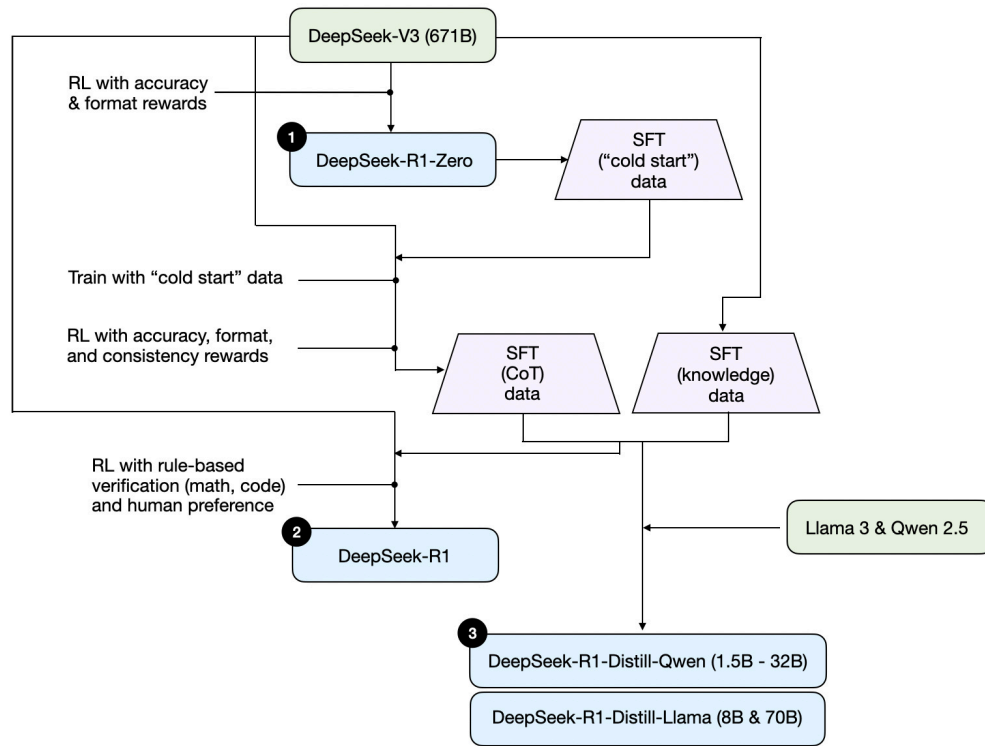
Case Study: The DeepSeek-R1 Family

DeepSeek released **three distinct models**, each teaching a different lesson:

Model	Lesson
DeepSeek-R1-Zero	Reasoning can emerge from pure RL , no SFT
DeepSeek-R1	SFT + RL produces the strongest model
DeepSeek-R1-Distill	Smaller models catch up via SFT distillation

All three are built on top of the [DeepSeek-V3 671B](#) base model. The [technical report](#) (or [arXiv](#)) is freely available and serves as a blueprint for the field.

The DeepSeek Training Pipeline (Bird's-Eye View)



All three models start from the same DeepSeek-V3 671B base. The recipes diverge — and each teaches a different lesson. We'll walk through each branch in turn.

The 4 Ways to Build a Reasoning Model

1. **Inference-time scaling** — no retraining
2. **Pure RL** — no SFT before RL
3. **SFT + RL** — the flagship recipe
4. **Pure SFT / distillation** — small, efficient models

These are *complementary*, not mutually exclusive.

Approach 1 — Inference-Time Scaling

Idea: spend more *compute at inference* to get better answers. No weight updates. Analogy: humans give better answers when given more time.

Three common flavors:

- **Chain-of-thought (CoT) prompting** — "think step by step"
- **Majority voting / self-consistency** — sample N answers, pick most common
- **Search-based decoding** — beam search, MCTS, process reward models

Regular prompting

Q: A juggler can juggle 16 balls. Half of the balls are golf balls, and half of the golf balls are blue. How many blue golf balls are there?

A: The answer (arabic numerals) is

(Output) 8 ✗

Chain-of-thought prompting

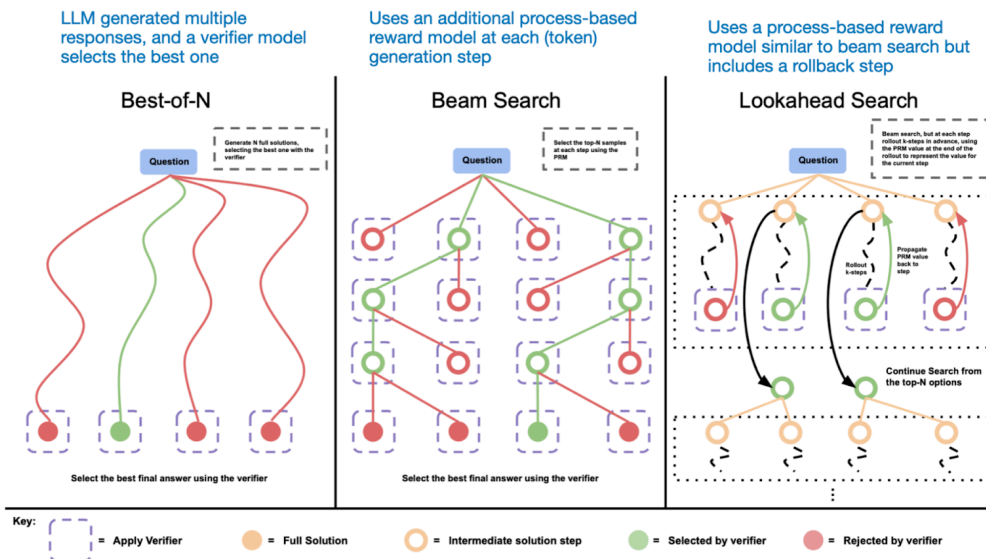
Q: A juggler can juggle 16 balls. Half of the balls are golf balls, and half of the golf balls are blue. How many blue golf balls are there?

A: **Let's think step by step.**

(Output) *There are 16 balls in total. Half of the balls are golf balls. That means that there are 8 golf balls. Half of the golf balls are blue. That means that there are 4 blue golf balls.* ✓

Adding "*Let's think step by step*" shifts the model into a different response distribution — often turning wrong answers into right ones. (Kojima et al., 2022)

Search-Based Inference Scaling



A **process reward model (PRM)** scores partial reasoning chains. Common variants:

- **Best-of-N** — sample N answers, pick the highest-scored
- **Beam search with PRM** — keep top-k partial chains at each step
- **MCTS** — explore branching trees of reasoning

See [Snell et al., *Scaling LLM Test-Time Compute Optimally* \(2024\)](#).

A Surprising Finding from DeepSeek

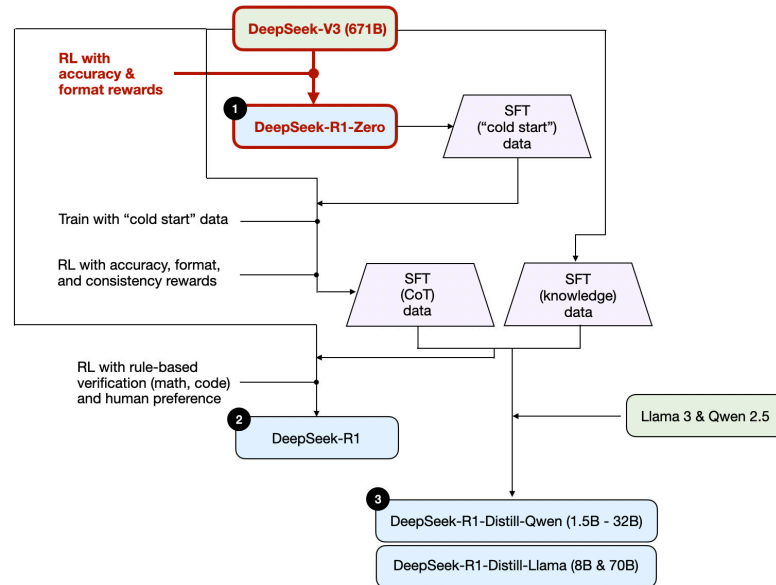
The DeepSeek-R1 report puts PRM-based and MCTS-based approaches under "**unsuccessful attempts**".

Their finding:

- Explicit search didn't help them much
- But R1 *naturally* produces longer reasoning chains than V3
- This is an **implicit** form of inference-time scaling

Caveat: explicit inference-time scaling often lives at the *application layer* (e.g. a wrapper around the model), not inside the weights. It is widely suspected that OpenAI's o1/o3 use some form of it — which would help explain why they are pricey per token.

Approach 2 — Pure Reinforcement Learning



The headline finding of DeepSeek-R1:

Reasoning can emerge as a learned behavior from RL alone, with no SFT warm-start.

Recipe for **R1-Zero**: start from DeepSeek-V3 base, skip SFT entirely, apply RL with *rule-based* rewards. This is unusual — standard RLHF has an SFT stage first.

The Reward Design for R1-Zero

Two rule-based rewards — no human preference model needed:

Accuracy reward

- Code → compile and run against tests (LeetCode-style)
- Math → deterministic verification of final answer

Format reward

- An LLM judge checks that responses wrap their reasoning in `<think> ... </think>` tags
- Enforces a structural convention, not a style preference

Key point: when the task has a *verifiable* answer, rule-based rewards sidestep the reward-modeling bottleneck of RLHF.

The "Aha!" Moment

Question: If $a > 1$, then the sum of the real solutions of $\sqrt{a - \sqrt{a+x}} = x$ is equal to

Response: <think>

To solve the equation $\sqrt{a - \sqrt{a+x}} = x$, let's start by squaring both ...

$$\left(\sqrt{a - \sqrt{a+x}}\right)^2 = x^2 \implies a - \sqrt{a+x} = x^2.$$

Rearrange to isolate the inner square root term:

$$(a - x^2)^2 = a + x \implies a^2 - 2ax^2 + (x^2)^2 = a + x \implies x^4 - 2ax^2 - x + (a^2 - a) = 0$$

...

Wait, wait. Wait. That's an aha moment I can flag here.

Let's reevaluate this step-by-step to identify if the correct sum can be ...

We started with the equation:

$$\sqrt{a - \sqrt{a+x}} = x$$

First, let's square both sides:

$$a - \sqrt{a+x} = x^2 \implies \sqrt{a+x} = a - x^2$$

Next, I could square both sides again, treating the equation: ...

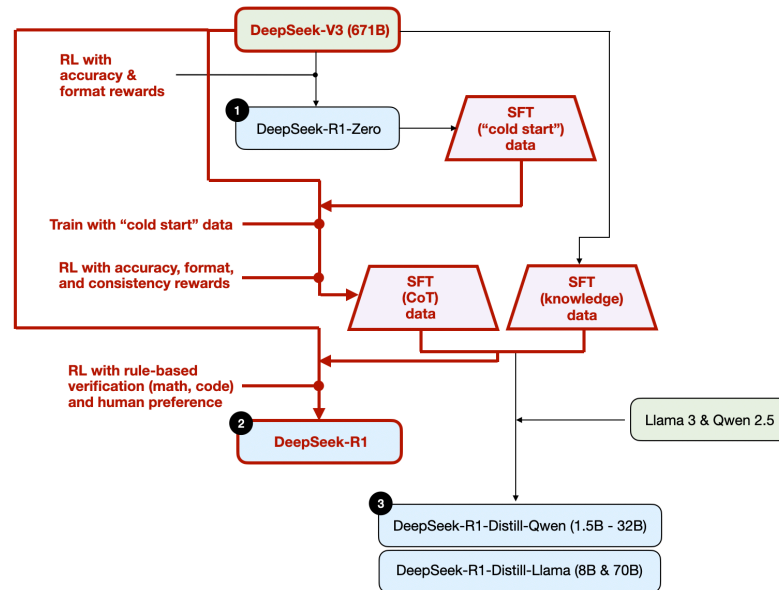
...

Table 3 | An interesting "aha moment" of an intermediate version of DeepSeek-R1-Zero. The model learns to rethink using an anthropomorphic tone. This is also an aha moment for us, allowing us to witness the power and beauty of reinforcement learning.

During R1-Zero training, the model spontaneously began to **pause, reconsider, and rewrite** its approach mid-answer. Nobody trained it to do that — the behavior emerged from reward alone.

R1-Zero isn't the strongest reasoning model — but it is the **proof of concept** that reasoning can be *grown*, not just *taught*.

Approach 3 — SFT + RL (the Flagship Recipe)



This is how you get **DeepSeek-R1** — and probably o1 and friends. Four stages:

1. Use R1-Zero to generate "cold-start" SFT data → instruction-tune V3
2. RL stage (accuracy + format + **language-consistency** reward)
3. Collect new SFT data: 600K CoT examples + 200K general examples
4. Final RL stage with *both* verifiable rewards and human-preference rewards

Why Add a Language-Consistency Reward?

A quirky failure mode of reasoning-trained LLMs:

Language mixing — the model starts answering in English, then drifts into Chinese mid-chain-of-thought, then back.

Why? Because the model's best reasoning path in latent space may not respect language boundaries.

Fix: a reward term that penalizes mixing languages within a single response.

Minor-looking detail, major UX improvement.

R1 vs. R1-Zero: Does the Extra Work Pay Off?

Model	Math benchmarks			Bio, physics & chemistry	Code benchmarks	
	AIME 2024		MATH-500	GPQA Diamond	LiveCode Bench	CodeForces
	pass@1	cons@64	pass@1	pass@1	pass@1	rating
OpenAI-o1-mini	63.6	80.0	90.0	60.0	53.8	1820
OpenAI-o1-0912	74.4	83.3	94.8	77.3	63.4	1843
DeepSeek-R1-Zero	71.0	86.7	95.9	73.3	50.0	1444
DeepSeek-R1	79.8		97.3	71.5	65.9	2029

Higher is better

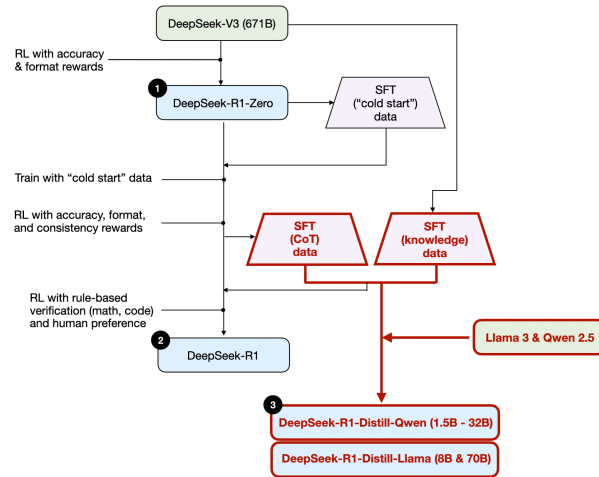
RL only →

SFT + RL →

SFT + RL consistently beats pure RL on challenging benchmarks. The RL-only model (R1-Zero) is scientifically more interesting; the SFT+RL model (R1) is practically more useful.

R1 is also broadly competitive with o1 — and substantially cheaper at inference time.

Approach 4 — Pure SFT (Distillation)



DeepSeek also shipped **R1-Distill** models: Qwen and Llama bases, sizes 1.5B–70B, fine-tuned on outputs from R1.

⚠ This is **not classical knowledge distillation** (no logit matching).
It's instruction fine-tuning on R1-generated answers.

Two reasons: **efficiency** (7B/14B run on a laptop) and **research signal** (how far does pure SFT get you?).

Distillation Results (R1-Distill)

Model	AIME 2024		MATH-500	GPQA Diamond	LiveCode Bench	CodeForces
	pass@1	cons@64	pass@1	pass@1	pass@1	rating
GPT-4o-0513	9.3	13.4	74.6	49.9	32.9	759
Claude-3.5-Sonnet-1022	16.0	26.7	78.3	65.0	38.9	717
OpenAI-o1-mini	63.6	80.0	90.0	60.0	53.8	1820
QwQ-32B-Preview	50.0	60.0	90.6	54.5	41.9	1316
DeepSeek-R1-Distill-Qwen-1.5B	28.9	52.7	83.9	33.8	16.9	954
DeepSeek-R1-Distill-Qwen-7B	55.5	83.3	92.8	49.1	37.6	1189
DeepSeek-R1-Distill-Qwen-14B	69.7	80.0	93.9	59.1	53.1	1481
DeepSeek-R1-Distill-Qwen-32B	72.6	83.3	94.3	62.1	57.2	1691
DeepSeek-R1-Distill-Llama-8B	50.4	80.0	89.1	49.0	39.6	1205
DeepSeek-R1-Distill-Llama-70B	70.0	86.7	94.5	65.2	57.5	1633
DeepSeek-R1-Zero	71.0		95.9	73.3	50.0	1444
DeepSeek-R1	79.8		97.3	71.5	65.9	2029

The distilled models are surprisingly strong:

- **R1-Distill-Qwen-32B** \approx matches o1-mini on many reasoning benchmarks
- **R1-Distill-Llama-70B** approaches full R1 on some tasks
- Far cheaper to serve than the 671B R1

Plausible interpretation: o1-mini may itself be a distilled version of o1.

Tradeoff: distillation needs a *stronger teacher model to already exist*.

A Revealing Experiment: Pure RL on a Small Model

Model	AIME 2024		MATH-500	GPQA Diamond	LiveCodeBench
	pass@1	cons@64	pass@1	pass@1	pass@1
pure RL → QwQ-32B-Preview	50.0	60.0	90.6	54.5	41.9
DeepSeek-R1-Zero-Qwen-32B	47.0	60.0	91.6	55.0	40.2
DeepSeek-R1-Distill-Qwen-32B	72.6	83.3	94.3	62.1	57.2
pure RL → DeepSeek-R1-Zero	71.0		95.9	73.3	50.0
DeepSeek-R1	79.8		97.3	71.5	65.9

DeepSeek also tried applying R1-Zero's pure-RL recipe to **Qwen-32B** directly. Result: worse than R1-Distill-Qwen-32B.

Interpretation:

- **Pure RL** is effective when the base model is already large/strong enough
- **Pure SFT on high-quality reasoning data** is more effective at smaller scales

Small base model → distill. Strong base model → RL.

Summary: When to Use Which Approach

Approach	Strength	Weakness
1. Inference-time scaling	No retraining, immediate gains	Ongoing per-query cost
2. Pure RL	Research insights, clean signal	Needs strong base model
3. SFT + RL	Best overall quality	Most expensive pipeline
4. Distillation (SFT)	Cheap, efficient, deployable	Ceilinged by teacher quality

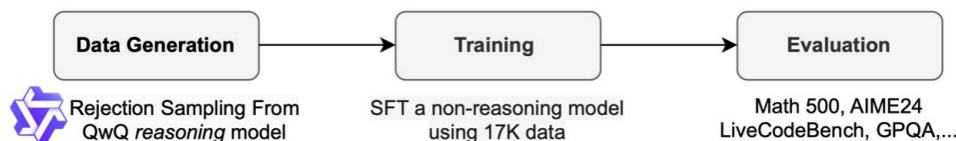
Practice: most production systems combine **SFT + RL (training)** with **inference-time scaling (serving)**.

Reasoning Models on a Budget

You do not need millions of dollars to play with these ideas.

Sky-T1 — \$450 to Train an o1-Preview-Level Model

“Alpaca” moment for reasoning data?



<https://novasky-ai.github.io/posts/sky-t1/>

\$450 in GPU costs

From NovaSky (UC Berkeley), Jan 2025:

- **Base model:** Qwen2.5-32B
- **Data curation via rejection sampling:** generate many candidate traces from a strong teacher, keep only the ones whose final answer verifies (math / code tests), SFT on the survivors.
- **Technique:** SFT only, no RL
- **SFT dataset:** just **17K examples** (generated from a stronger model)
- **Total training cost:** ~\$450 — less than an AI conference registration
- **Result:** matches o1-preview on several reasoning benchmarks

Moral: high-quality, well-curated data > lots of data.

TinyZero — Pure RL for \$30

User: Using the numbers [19, 36, 55, 7], create an equation that equals 65.

Assistant: Let me solve this step by step.

<think> We need to find an equation using the numbers 19, 36, 55, and 7 exactly once, with basic arithmetic operations, that equals 65. One possible combination is $55 + 36 - 19 + 7$. Let's check: $55 + 36 = 91$, $91 - 19 = 72$, and $72 + 7 = 79$. However, this doesn't equal 65. Another combination is $55 + 36 + 7 - 19$. Let's check: $55 + 36 = 91$, $91 + 7 = 98$, and $98 - 19 = 79$. This also doesn't equal 65. After trying different combinations, I found that $55 + 36 + 7 - 19 = 79$, which is close but not equal to 65. Finally, I found that $55 + 36 - 19 + 7 = 79$, which is still not equal to 65. However, if we try $55 + 36 - 7 + 19$, we get $55 + 36 = 91$, $91 - 7 = 84$, and $84 + 19 = 103$, which is not equal to 65. After further trial and error, I found that $55 + 36 + 7 - 19 = 79$, which is still not equal to 65. However, if we try $55 + 36 - 7 - 19$, we get $55 + 36 = 91$, $91 - 7 = 84$, and $84 - 19 = 65$. This equals 65. </think>

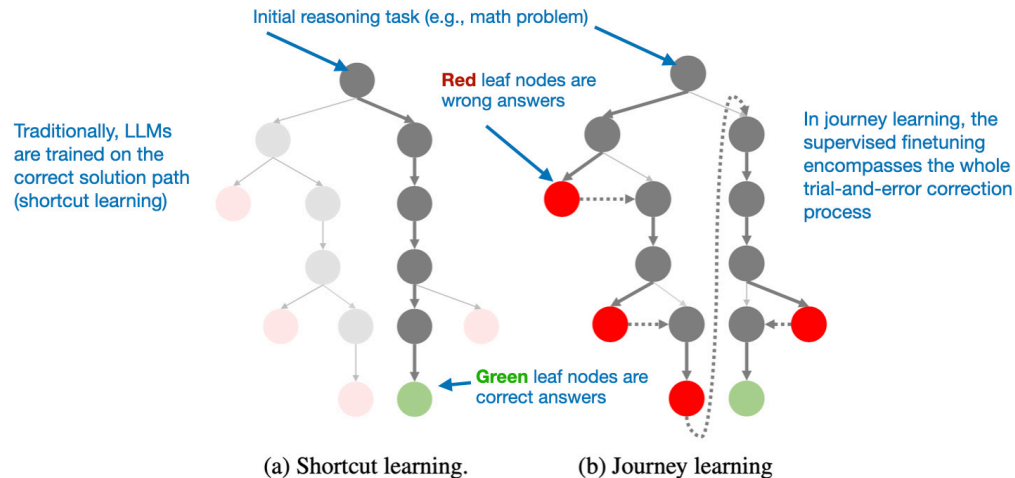
<answer> $55 + 36 - 7 - 19$ </answer>

From Berkeley, Jan 2025:

- **Base model:** Qwen2.5-3B
- **Technique:** pure RL (R1-Zero-style) on the *Countdown* number puzzle
- **Total training cost:** <\$30
- **Observation:** even this small model developed **self-verification** behavior

Shows the R1-Zero finding replicates at very small scale, on a narrow domain.
Not a general reasoner — but a powerful proof of concept.

Journey Learning — A Twist on SFT



From [O1 Replication Journey — Part 1 \(Oct 2024\)](#):

Shortcut learning (standard SFT): train only on *correct* paths — model learns "what a good answer looks like."

Journey learning: train on *both correct and incorrect* paths with reflection — "I tried X, that didn't work because Y, so I tried Z."

Resembles the self-correction that emerges in RL — but achievable with pure SFT.

Prompting Reasoning Models — Quick Tips

Reasoning models behave differently from regular LLMs. From the R1 paper and community findings:

1. **Zero-shot beats few-shot** — few-shot examples often *hurt* reasoning models
2. **Describe the problem directly** — skip elaborate prompting patterns
3. **Keep language consistent** — mixing languages in the prompt causes mixing in the output
4. **Let it think** — don't ask for "short" answers on hard problems

Your elaborate prompt-engineering tricks for GPT-3.5 may actively *hurt* o1 and R1.

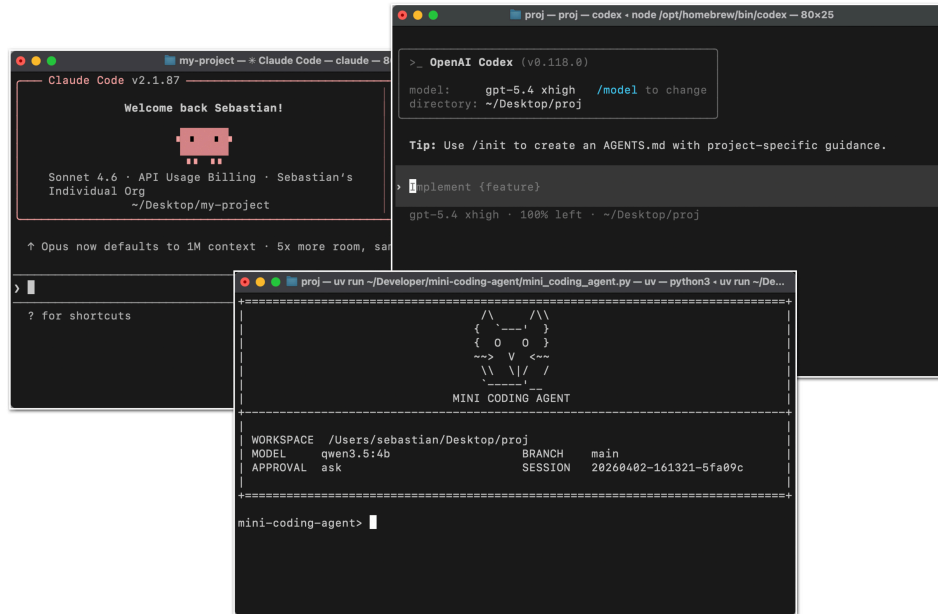
Part 1 Takeaways

- "Reasoning LLM" \approx LLM specialized for multi-step, verifiable tasks
- Four complementary techniques: **inference-time scaling, pure RL, SFT+RL, distillation**
- **SFT + RL** is the flagship recipe (DeepSeek-R1, probably o1)
- **Reasoning can emerge** from pure RL with rule-based rewards (R1-Zero's headline finding)
- **Distillation** is the most cost-effective path for smaller models
- Budget-friendly variants exist: **Sky-T1 (\$450), TinyZero (<\$30)**

Now: what happens when we put one of these models inside an **agent**?

Part 2: Coding Agents

Why Talk About Agents?



In 2024–2025, a lot of practical LLM progress was **not** about better models. It was about **better scaffolding around the models**: tool use, context management, memory, control loops.

This is why **Claude Code** or **Codex CLI** can feel dramatically more capable than the *same model* in a plain chat interface.

The model is the engine. The harness is the car.

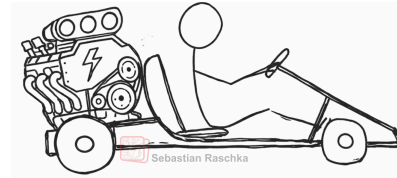
LLM vs. Reasoning Model vs. Agent



Conventional LLM



Reasoning LLM
(more expensive to run)



LLM in agent harness

Concept

What it is

LLM

A next-token model

Reasoning model

An LLM optimized to produce intermediate reasoning and verify itself

Agent

A control loop that uses a model + **tools** + **memory** + **environment feedback**

Agent harness

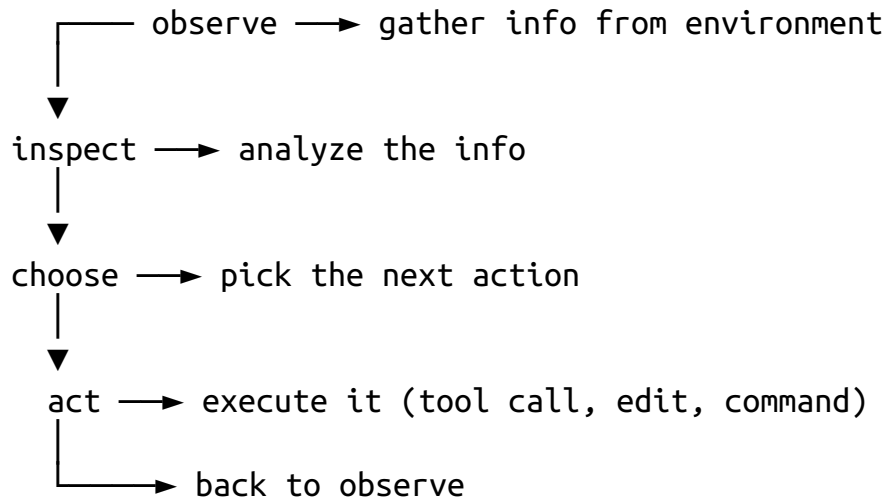
Software scaffold around the agent (prompts, tools, state, control flow)

Coding harness Task-specific harness for software engineering

Claude Code and Codex CLI are **coding harnesses**.

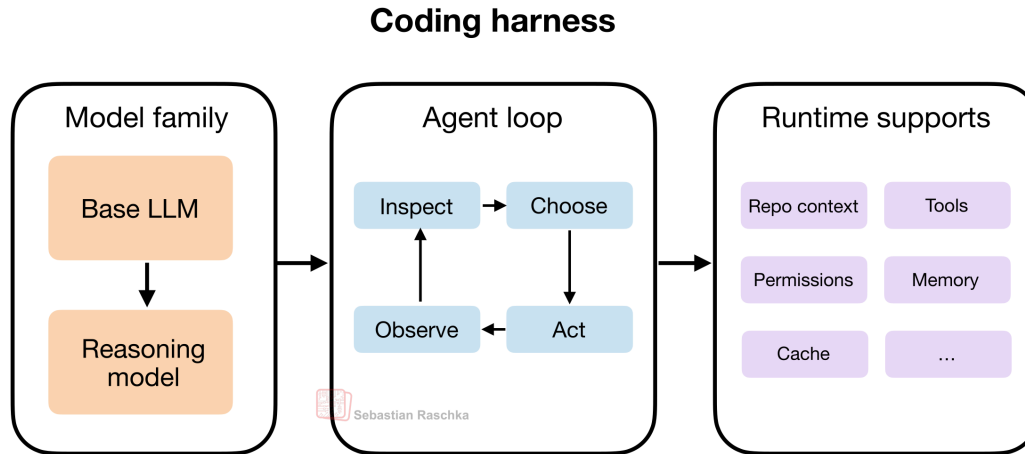
The Agent Loop

An agent repeatedly runs a loop inside an environment:



Stop when the goal is met (or budget/limit is reached).

A Coding Harness Has Three Layers



1. **Model family** — the LLM / reasoning model (the engine)
2. **Agent loop** — observe → inspect → choose → act
3. **Runtime supports** — repo context, tools, caching, memory, sandboxing

Same model + better harness = dramatically different UX.

The Claim: Harness > Model (Often)

Today's frontier LLMs from different vendors have roughly comparable raw capability.

Raschka's claim:

If you drop a strong open-weight model (say GLM-5) into the same harness as Codex, it will likely perform on par with GPT-5.4 in Codex or Claude Opus 4.6 in Claude Code.

The harness is frequently the distinguishing factor.

(Small caveat: there is usually some *harness-specific post-training*, e.g. OpenAI maintains gpt-5.3 and gpt-5.3-codex variants.)

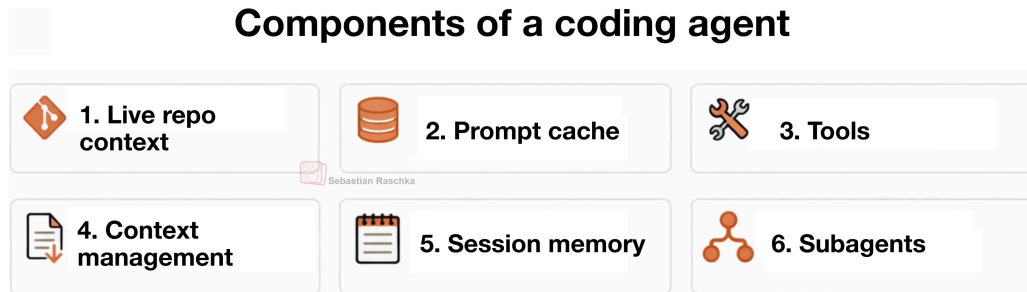
Coding Is Not Just "Writing Code"

Why coding needs a harness more than most tasks:

- **Repo navigation** — thousands of files, which ones matter?
- **Function lookup** — where is `encode_batch` defined?
- **Diff application** — edit existing code, don't rewrite files
- **Test execution** — run tests, read output, fix failures
- **Error inspection** — parse tracebacks, map to source
- **Iterative feedback** — try, observe, revise

Next-token prediction alone doesn't solve these. A harness does.

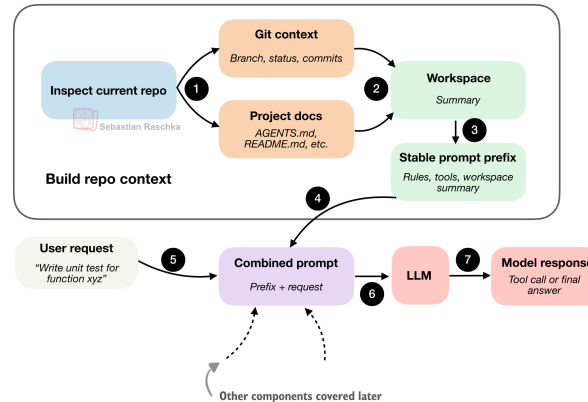
The Six Components of a Coding Harness



- 1. Live repo context**
- 2. Prompt shape and cache reuse**
- 3. Structured tools, validation, and permissions**
- 4. Context reduction and output management**
- 5. Transcripts, memory, and resumption**
- 6. Delegation with bounded subagents**

We'll walk through each.

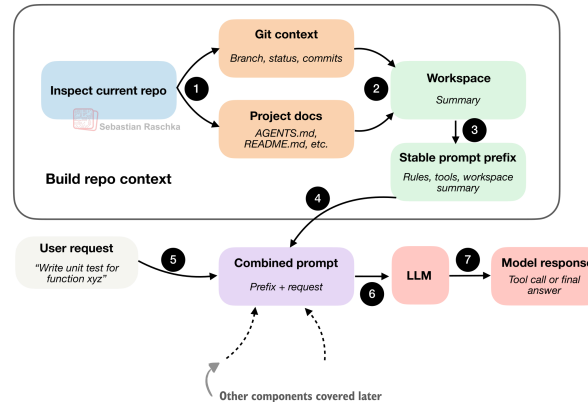
Component 1 — Live Repo Context



When the user says *"fix the tests"*, the model cannot answer in a vacuum. The harness must know:

- Git repo? Branch? Uncommitted changes?
- Is there an AGENTS.md / CLAUDE.md / README?
- Repo layout? What build/test commands are defined?

Component 1 — Live Repo Context



When the user says *"fix the tests"*, the model cannot answer in a vacuum. The harness must know:

- Git repo? Branch? Uncommitted changes?
- Is there an `AGENTS.md` / `CLAUDE.md` / `README`?
- Repo layout? What build/test commands are defined?

Without it: *"I'll run `pytest`"* — wrong, this project uses `make test`. **With it:** *"`AGENTS.md` says tests run via `make test`. Running now..."*

This is why files like `AGENTS.md` and `CLAUDE.md` have become a de facto ecosystem convention.

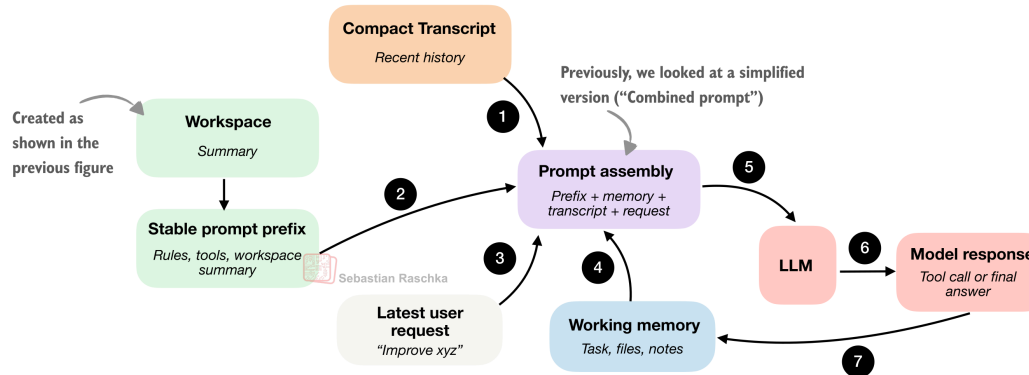
Component 2 — Prompt Shape and Cache Reuse

Naive approach: on every turn, concatenate everything and send it to the model.

Problem: **most of that prompt doesn't change between turns.**

- System instructions — stable
- Tool descriptions — stable
- Workspace summary — mostly stable
- Short-term memory — changes
- Recent transcript — changes
- Latest user turn — changes every time

Split Into "Stable Prefix" + "Dynamic Tail"



The **stable prefix** contains system instructions, tool descriptions, and workspace summary — all mostly unchanging across turns.

The **dynamic section** contains short-term memory, recent transcript, and the new user request.

Smart runtimes **cache the Key-Value weights for the prefix** → massive latency and cost wins on long coding sessions. Anthropic and OpenAI both expose prompt caching APIs for exactly this reason.

What is Prompt Caching?

Problem: every API call re-processes the full prompt through all attention layers — expensive for long system prompts, tool schemas, and growing agent transcripts.

What is Prompt Caching?

Problem: every API call re-processes the full prompt through all attention layers — expensive for long system prompts, tool schemas, and growing agent transcripts.

Idea: the provider caches the **KV (key/value) tensors** computed for a prompt *prefix* on their servers. Identical prefixes on later calls skip the prefill and load cached state.

What is Prompt Caching?

Problem: every API call re-processes the full prompt through all attention layers — expensive for long system prompts, tool schemas, and growing agent transcripts.

Idea: the provider caches the **KV (key/value) tensors** computed for a prompt *prefix* on their servers. Identical prefixes on later calls skip the prefill and load cached state.

- **Prefix-only, exact-token match** — one character change invalidates the rest
- **TTL-based eviction** (Anthropic: 5 min or 1 hr; OpenAI: ~5–10 min idle)
- **Pricing:** writes cost *more* (~1.25–2x), reads cost *much less* (~0.1–0.5x)

Prompt Caching in Practice

Providers:

- **Anthropic** — explicit `cache_control` breakpoints (up to 4)
- **OpenAI** — automatic for prompts ≥ 1024 tokens
- **Google Gemini** — explicit `CachedContent` handle

Why it's critical for coding agents:

- Agents replay a *growing* transcript on every tool-call iteration
- Large, stable system prompts + tool schemas + file context
- Without caching: a 20-step agent task reprocesses the full prompt 20×
- With caching: each turn only prefills the new tool result → **5–10× lower latency, ~10× lower cost**

Design rule: put **stable content first** (system prompt → tools → docs → history → new user turn). Never interpolate timestamps or request IDs near the top.

Component 3 — Structured Tools

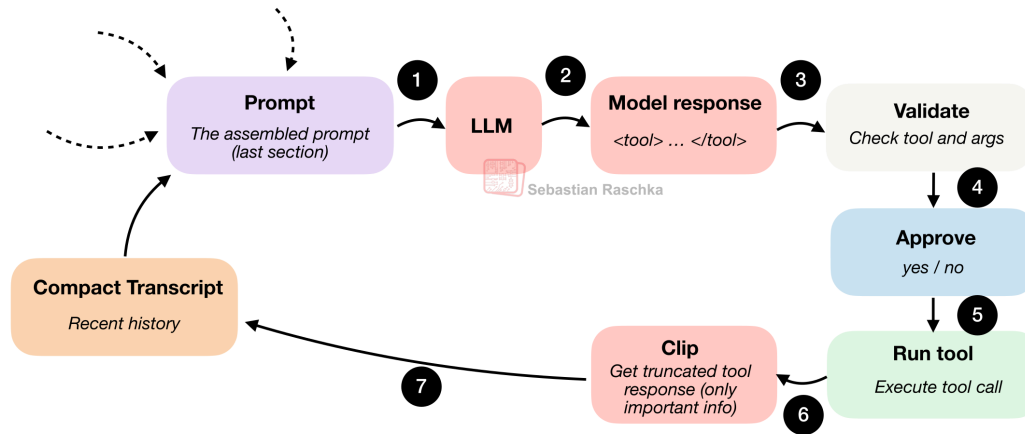
A plain LLM can *suggest* shell commands in prose:

"You could run `pytest tests/test_auth.py -v`"

A coding agent *actually runs them* — and uses the output. But not via free-form generation. It uses **structured tool calls**:

- `list_files(path)`
- `read_file(path, start, end)`
- `search(pattern, path)`
- `run_shell(cmd)`
- `write_file(path, contents)`
- `apply_diff(path, diff)`

The Tool-Use Flow



1. Model emits a **structured action**
2. Harness **validates**: known tool? valid args? path inside workspace? needs approval?
3. (If needed) prompt user for **approval**
4. **Execute**, capture output
5. Feed **clipped result** back into the loop

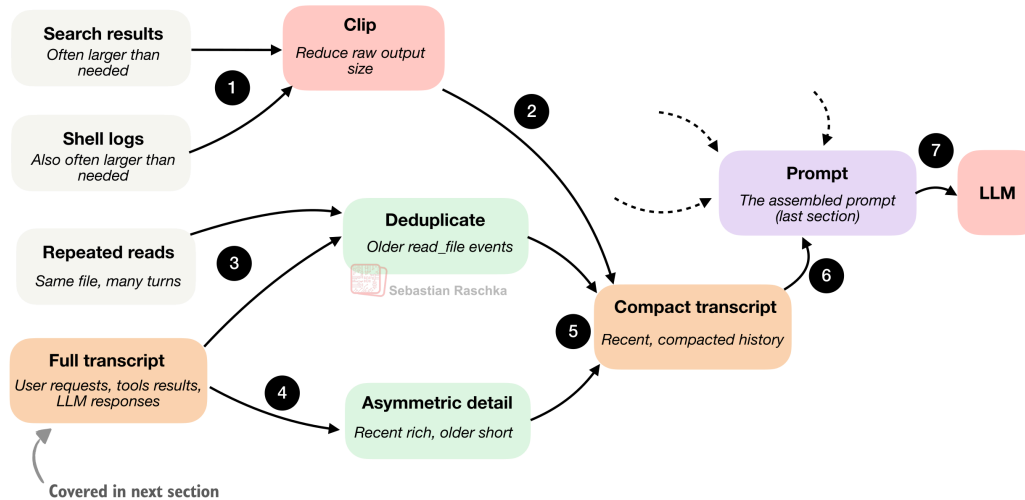
Component 4 — Minimizing Context Bloat

Coding agents are context-hogs. Every step can produce:

- A file read (hundreds of lines)
- A tool output (stack traces, logs)
- A search result (many hits)
- A diff

Left unchecked, the context fills up in a few turns. Even with million-token context windows, performance degrades on long contexts (and cost scales with tokens).

Two Compaction Strategies



Clipping — truncate long outputs (file reads, test logs, search results); show head/tail, elide middle. Prevents any one thing from dominating the budget.

Transcript reduction — turn full session history into a smaller summary.

- **Recency-weighted:** keep recent events richer, compress older ones more
- **Deduplicate repeat file reads** — don't show the same file 5 times

Context Quality > "Model Quality" (Often)

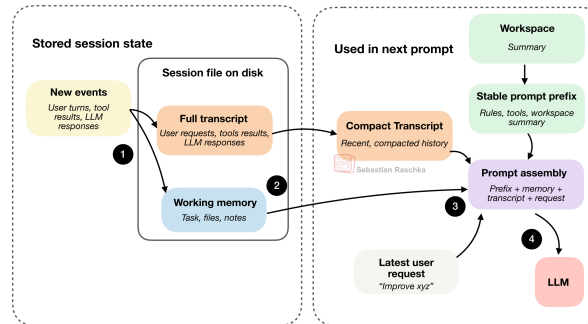
Raschka's aside:

"A lot of apparent 'model quality' is really context quality."

If you've ever watched Claude Code or Cursor repeatedly re-read the same file, or fail to recall something from 20 turns ago — that's a context-management problem, not usually a model problem.

Good context engineering is the "boring" infrastructure work that separates great coding agents from mediocre ones.

Component 5 — Structured Session Memory



Two layers of state, with different jobs:

Layer	Format	Role
Full transcript	JSONL, append-only	Durable record. Resumable if agent crashes.
Working memory	JSON, structured	Distilled state: current task, key files, recent notes. Updated and compacted, not just appended.

Compact transcript (Component 4) ≠ working memory:

- **Compact transcript** → for prompt reconstruction
- **Working memory** → for task continuity

What Goes Into Working Memory?

Examples of structured working-memory fields:

```
{
  "current_task": "fix failing auth tests",
  "key_files": ["src/auth.py", "tests/test_auth.py"],
  "recent_notes": [
    "test_login_timeout fails due to missing mock",
    "auth.py line 142 uses deprecated API"
  ],
  "open_questions": ["should we keep backward compat?"]
}
```

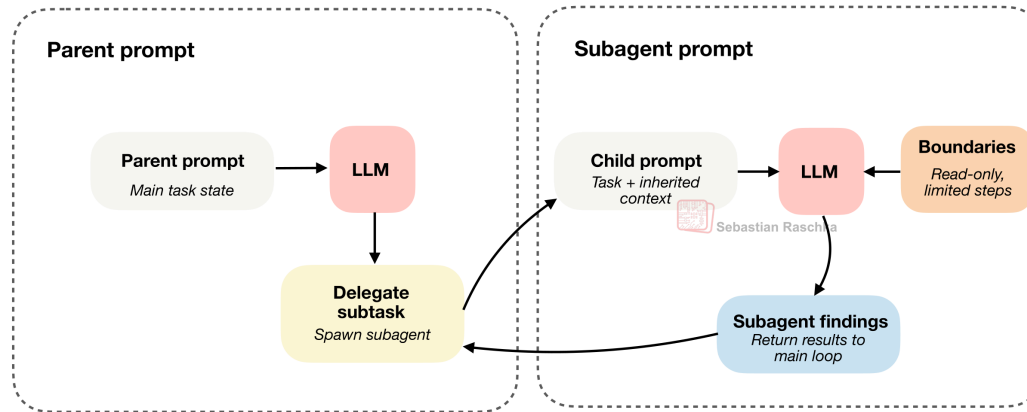
Small, curated, editable. Not a dump of everything that happened.

Why Separate Memory from Transcript?

- **Transcripts** are honest but noisy and linear
- **Memory** is curated and queryable
- **Transcript** = "what happened"
- **Memory** = "what matters right now"

Close your laptop, reopen tomorrow → the agent loads both. The transcript grounds it; the working memory reminds it what it was doing.

Component 6 — Delegation via Subagents



Sometimes the main agent needs a side answer:

- "Which file defines `encode_batch`?"
- "What does this config say?"
- "Why is this test failing?"

Options:

- **Inline:** do it in the main loop → pollutes context, may lose focus
- **Delegate:** spawn a bounded **subagent** → parallel, isolated work

Claude Code has supported subagents for a while; Codex added them more recently.

Binding the Subagent

"The tricky design problem is not just how to spawn a subagent, but how to bind one."

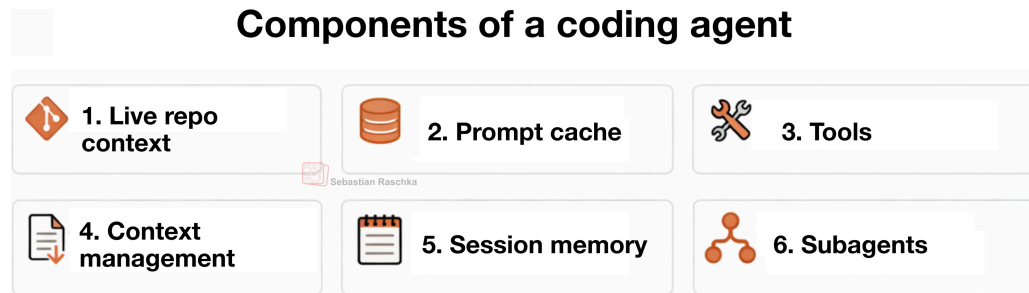
A subagent needs **enough context to be useful** — but must be **constrained** or it'll duplicate work, spawn its own subagents, edit files it shouldn't, or run forever.

Typical constraints:

- **Read-only by default** (sometimes — Codex is more permissive)
- **Scoped context**: inherits relevant parts of parent's state
- **Recursion depth limit**
- **Tool whitelist** — smaller than parent's
- **Timeout / step budget**
- **Cannot spawn its own subagents** (usually)

Return a focused answer to the parent, then disappear.

The Six Components at a Glance



#	Component	Why it matters
1	Live repo context	Grounds ambiguous instructions in concrete facts
2	Prompt shape + caching	Saves cost & latency on repeated state
3	Structured tools + permissions	Deterministic, safe, auditable actions
4	Context reduction	Keeps sessions long without drowning in tokens
5	Transcripts + memory	Durability + task continuity
6	Bounded subagents	Parallel side-tasks without losing focus

Together, they turn "an LLM that writes code" into "a system that ships PRs."

Coding Harness ≠ General Agent Harness

Not every agent is a coding agent.

Coding harness (Claude Code, Codex) — optimized for:

- one developer, one repo at a time
- tight feedback with files, tools, tests
- minutes-to-hours sessions

General agent platform (e.g. OpenClaw) — optimized for:

- many long-lived agents
- across chats, channels, and workspaces
- coding is *one* workload among many

Both use similar building blocks. The emphasis differs.

Putting It All Together

How Part 1 and Part 2 Connect

Part 1: how do we **build** a model that can reason well? Part 2: how do we **deploy** that model in a system that actually ships useful work?

Neither is sufficient alone:

- A great reasoning model in a bad harness → feels dumb
- A great harness around a weak model → feels stuck

The state of the art in 2026: **strong reasoning model** × **strong coding harness** = what we see in Claude Code, Codex, Cursor.

Key Takeaways

1. **Reasoning is a specialization**, not a replacement. Use it when tasks need multi-step verification.
2. **SFT + RL** is the flagship recipe; **distillation** is the budget path; **inference-time scaling** is the serving-side boost.
3. **Rule-based rewards** (accuracy + format) can replace human preference models when answers are verifiable.
4. **A harness can matter as much as the model**. Coding is partly about next-token prediction, but mostly about managing context, tools, and state.
5. **Context quality is often mistaken for model quality**. Clipping, deduplication, caching, and working memory do a lot of the heavy lifting.
6. **Budget-friendly research is possible**: Sky-T1 (\$450), TinyZero (<\$30), and the Mini Coding Agent are all doable at lab scale.

Further Reading

Reasoning models

- DeepSeek-R1 technical report — arxiv.org/abs/2501.12948
- Snell et al., *Scaling LLM Test-Time Compute Optimally* — arxiv.org/abs/2408.03314
- *O1 Replication Journey — Part 1* — arxiv.org/abs/2410.18982
- Sky-T1 blog — novasky-ai.github.io/posts/sky-t1/
- TinyZero repo — github.com/Jiayi-Pan/TinyZero

Coding agents

- Raschka's *Mini Coding Agent* — github.com/rasbt/mini-coding-agent
- Raschka, *Components of a Coding Agent* (Apr 2026)
- Raschka, *Build a Reasoning Model (From Scratch)* — Manning, 2026

Discussion Questions

1. Where would you expect a reasoning model to *fail* in ways a standard LLM wouldn't?
2. The R1-Zero "Aha!" moment — is this genuine emergence, or an artifact of the reward shape? How would you design an experiment to tell?
3. If harness > model for many tasks, what does that mean for *benchmarking*? Are current LLM leaderboards measuring the right thing?
4. For your own research/work: which of the six harness components would be hardest to reuse in a non-coding domain (e.g., a data-analysis agent)?
5. Where do you expect the next bottleneck to be — model capability, harness design, or something else entirely?

Thank You

Questions?

Slides and references available on the course page.