FOUNDATIONAL CONCEPTS: TECHNICAL BRIEF

Soar Chunking and Procedural Memory

Soar converts slow problem solving steps into fast production rules. Chunking performs the conversion in one automatic step. Current agents lack that step in 2026.

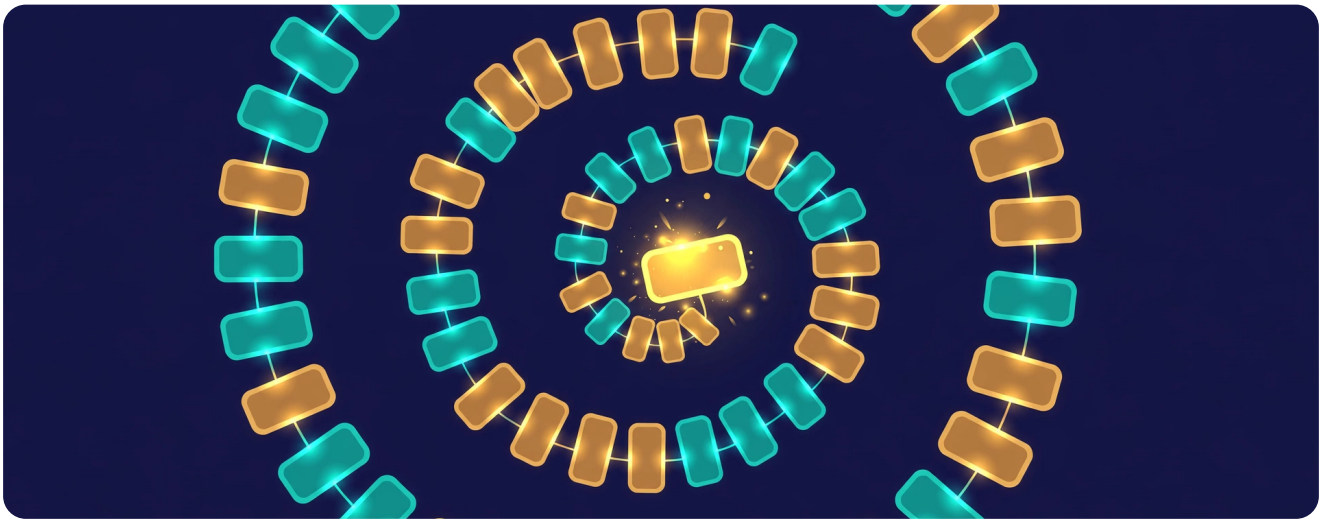

Figure 1: A long trace collapses into one stored rule.

Start Here: The Short Version

This method helps an AI turn slow problem solving into a fast, reusable skill. The system first works through a hard task step by step. The system then saves the path as one rule. The next similar task takes far fewer steps.

Think of a new worker on the first day of a job. The worker stops and thinks at each small step. After a few weeks, the worker knows the routine by heart. The worker does the job fast and with less effort. This method gives an AI a way to make the same jump. The AI keeps the lesson from hard work. The AI does not need to learn the lesson again.

WHAT IT CAN DO

- It watches a successful work path and finds the part that repeats.
- It saves that part as a skill with clear start conditions.
- It reuses the skill on later tasks and cuts wasted steps.

Each skill has a clear start point and a clear end result. A person can read the skill and test the skill.

WHERE PEOPLE CAN USE IT

- A support agent answers the same kind of request each day. The agent resets a password or checks an order status. A saved skill handles the repeat steps in the same safe way each time.
- A repair team sets up the same system or fixes the same fault often. A saved skill guides the repeat steps and leaves hard choices to people.
- A training system runs the same practice scene many times. The system learns from each run and gets faster and more steady over time.
- A warehouse robot meets a blocked aisle on a delivery run and hits an impasse. The robot opens a substate and solves the blockage step by step: reverse to the junction, scan the side aisle, reroute to the bin. Chunking converts the recovery trace into one production rule with start conditions for aisle width and blockage type. On the next blocked aisle, the rule fires at once and the robot reroutes with no new search.

PROBLEMS IT CAN HELP SOLVE

- Agents often think through the same task from zero every time. This habit wastes time and money and gives uneven results.
- Most agent skills today are text written by hand, not tested rules. A person writes each skill and hopes the model reads it in the right way.
- Teams lack a clear view of which skills work and which skills fail. Without a test record, a team has no firm base for fixes.

The rest of this brief gives the history, the method, the 2026 gap, and a first project plan.

1: Identification

John Laird, Allen Newell, and Paul Rosenbloom built Soar from 1983. The first full description appeared in 1987. Soar belongs to a class of systems with the name cognitive architectures.

John Anderson built ACT-R in the same period. ACT-R also splits memory into declarative memory and procedural memory. Teams used Soar in military simulations and in tutoring systems. Teams also used Soar in research on human performance.

Figure 2: Timeline. The procedural gap stays open in 2026.

2: Purpose

Soar solves problems in a problem space. A problem space holds states, operators, and a goal. An operator changes one state into a new state.

Soar searches the space for a path to the goal. The system learns from each problem at the same time. Soar combines problem solving and learning in one system.

3: Operation

THE DECISION CYCLE

1. The system reads the current state from working memory.
2. Production rules propose operators for the state.
3. Preferences rank the proposed operators.
4. The system selects one operator.
5. The selected operator changes the state.

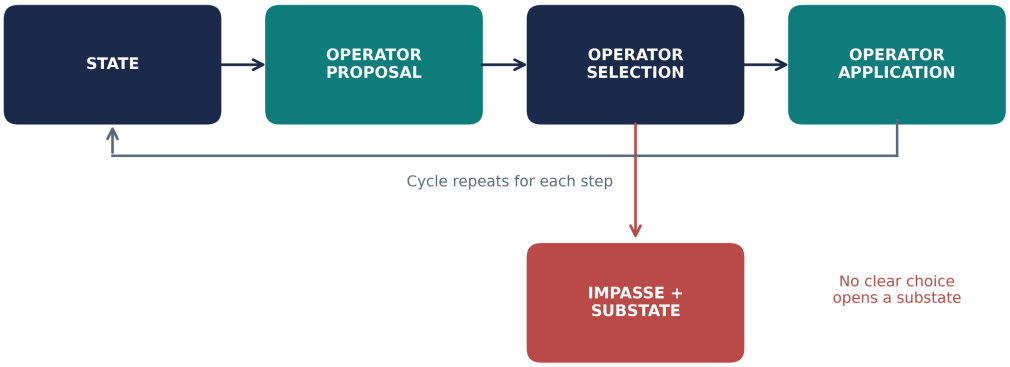

Figure 3: The decision cycle. A side path handles an impasse.

IMPASSE AND SUBSTATE

An impasse occurs at a step with no clear operator choice. Soar opens a substate for the impasse. Soar solves the substate as a separate problem. The substate solution ends the impasse.

3: Operation, continued

CHUNKING

1. Chunking examines the completed substate solution.
2. Chunking extracts the conditions and the result from the solution.
3. Chunking converts the solution into one new production rule.
4. The new rule enters procedural memory.
5. The new rule prevents the same impasse in later problems.

The rule fires at once in a later problem. The agent skips the slow substate search in that later problem. Chunking produces one fast skill from one slow trace.

> Chunking differs from a log. A log records the trace. Chunking converts the trace into a rule with conditions.

FOUR MEMORY TYPES

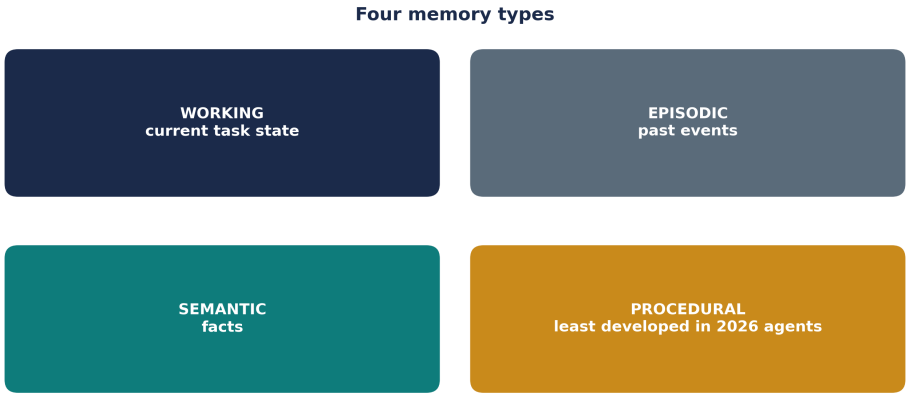

Figure 4: Four memory types. The amber quadrant marks the gap.

Memory type	Content	Status in 2026 agents
Working	Current task state	Strong
Episodic	Past events	Medium
Semantic	Facts	Strong
Procedural	Skills and rules	Weak

4: History

Experts wrote the production rules by hand. Each new domain needed new rules from an expert. The authoring cost blocked wide use.

Deep learning then took the main research problems in perception. Soar solved reasoning tasks well and perception tasks poorly. Research teams moved staff and funds to neural methods.

5: The 2026 Gap

Sumers and co-authors published the CoALA framework in 2023. CoALA maps language model agents to four memory types. The four types are working, episodic, semantic, and procedural.

CoALA and later work state a clear gap in procedural memory. Procedural memory receives the least development in current agents. Agent products store facts in semantic stores today. Agent products store past events in episodic logs today.

Skill libraries in agent products contain hand-written text today. A person still writes each skill by hand. The skill then stays as text for a model to read. No product compiles successful agent traces into tested rules in an automatic process.

> Products store facts and events. Products lack a compiler for skills.

6: Project Proposal

Goal. The desired end state is one trace compiler in daily use by an agent framework. The compiler reads successful task traces and returns tested skills. Agents load the skills at task start and skip the steps the skills cover.

Proof target. The project proves one claim on repeat tasks. Compiled skills cut task steps and completion time against hand-written skill text. Success rates stay equal or improve.

Product at the end. The product is the Skill Compiler and Registry, version 1. The product contains a trace reader, a subpath detector, a skill compiler, a trace test runner, and a versioned skill registry. Framework teams install the compiler beside the agent runtime.

The proposal builds a trace compiler for agent frameworks. The compiler applies the chunking idea to modern agent traces.

1. The compiler reads successful task traces from an agent.
2. The compiler finds repeated successful subpaths in the traces.
3. The compiler converts each subpath into a parameterized skill.
4. Each skill holds preconditions and steps in a clear form.
5. A test step runs each new skill against old traces.
6. A registry stores the skills with version numbers and success counts.
7. The agent loads skills from the registry at task start.

Figure 5: Trace compiler pipeline. The registry feeds the next task.

6: Project Proposal, continued

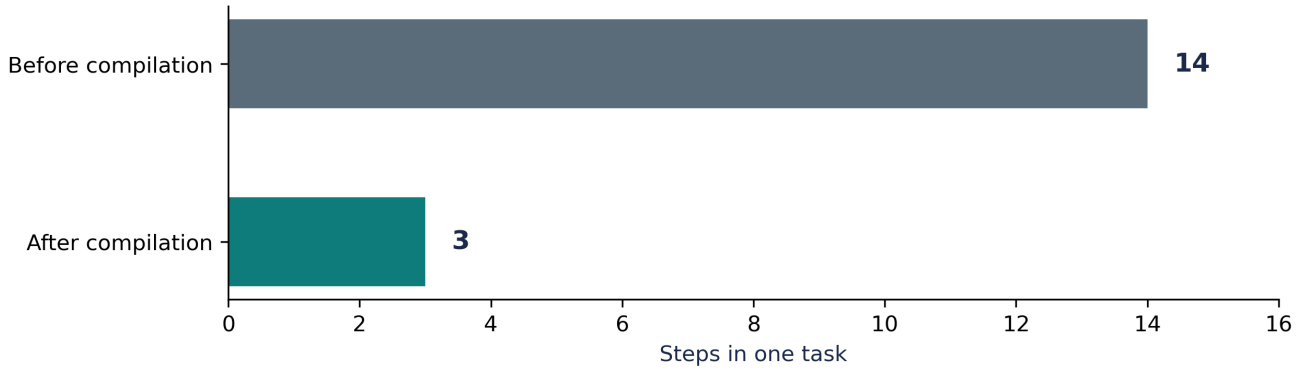

Figure 6: Example task. Steps drop from 14 to 3 after compilation.

One example task uses 14 steps before compilation. The same task uses 3 steps after compilation.

7: Risks and Limits

Bad traces produce bad skills. A skill with wrong preconditions fires in the wrong task. Tests on old traces miss new edge cases. A registry without version control hides the source of a failure.

Small tests on one narrow task reduce each risk. Human review of each new skill adds a final check.

> VERDICT: TIER 2
>
> Teams mine Soar for design ideas today. A small trace compiler tests the core idea in weeks. The project targets a gap with no product in 2026.