Society of Meta-Agents: The SOUL Motivation Framework for Specific and Objective Understanding Logic

Keyvan M. Sadeghi

April 22, 2025

1 Literature Review

We have identified six practical schools of agent motivation:

- **Intrinsic Motivation in RL**: Curiosity and surprise as auxiliary rewards to drive exploration in sparse environments (Schmidhuber, 2010; Pathak et al., 2017).
- **Information-Theoretic Drives**: Maximizing mutual information, novelty, and empowerment signals (Klyubin et al., 2005; Salge et al., 2014; Goertzel, 2024).
- **Competence-Progress Models**: Rewarding measurable learning progress toward self-generated subgoals to form a self-curriculum (Oudeyer & Kaplan, 2007).
- **Homeostatic/Drive-Reduction**: Minimizing internal variables (e.g., prediction error) via cybernetic drives (Hull, 1943; Friston, 2010).
- **Cognitive Architectures**: Embedding scalar drives into symbolic cycles (Soar's operator preferences; ACT-R's Expected Value of Control; Goertzel, 2024) for precise control updates.
- **Developmental-Robotics Hybrids**: Combining maturational constraints with competence progress to emulate developmental curricula (Lungarella et al., 2003).

2 Mathematical Foundations and Architectural Decisions

2.1 The Motivation Vector

At the heart of the SOUL Motivation Framework is the hidden internal state, the Motivation Vector:

$$\mathbf{s}_t = [s_c, s_u, s_h]$$

where s_c is competence, s_u is novelty/surprise, and s_h is homeostasis. This vector is updated after every interaction and drives all agent actions. In code, it is maintained as a Python dataclass and is never exposed to the LLM.

2.2 Motivation Vector Updates

- **Competence Progress:**

$$\Delta_c = p_g(t) - p_g(t-1), \quad s_c \leftarrow \mathrm{clip}(s_c + \alpha\Delta_c, 0, 1)$$

  where $p_g(t)$ is the agent's measured performance at time t.

- **Novelty/Surprise:**

$$\mathrm{novel}(t) = 1 - \frac{\mathbf{e}(t) \cdot \mu_{t-1}}{\|\mathbf{e}(t)\| \, \|\mu_{t-1}\|}, \quad s_u \leftarrow \mathrm{clip}(s_u + \alpha \, \mathrm{novel}(t), 0, 1)$$

  where $\mathbf{e}(t)$ is the current context embedding and μ_{t-1} is the rolling mean.

- **Homeostatic Decay:**

$$s_h \leftarrow (1-\delta)s_h + \delta$$

  This ensures s_h gently returns to baseline.

2.3 Meta-Graph and Rule Engine

The agent maintains a symbolic meta-graph G (a directed graph of rewrite rules R and meta-rules M). Each node represents a pattern or rule, and edges encode transformations or relationships. In Python, this is implemented with `networkx.DiGraph`.

2.4 Thresholded Nudge and Confidence

At each step, the agent computes a confidence score C_t:

$$C_t = f_{\mathrm{match}}(x_t, G, \mathbf{s}_t)$$

where f_{match} is a similarity or density function over meta-graph patterns and the current state. The agent compares C_t to a dynamic threshold τ_t:

$$\text{If} \quad C_t \geq \tau_t \quad \text{then nudge; else remain silent (null action)}$$

Concrete Example: If C_t is computed as the softmax of pattern matches in G weighted by $\mathbf{s}_t$, and τ_t is set adaptively (e.g., as a running quantile), then the agent only acts when it is sufficiently confident.

2.5 Perception–Cognition–Action Loop

The agent's operation at each time t is:

$$\begin{aligned}
\text{Perceive:} \quad & y_t = \mathrm{Perceive}(x_t) \\
\text{Update:} \quad & \mathbf{s}_{t+1} = \mathrm{UpdateMotivation}(\mathbf{s}_t, y_t) \\
\text{Record:} \quad & G_{t+1} = \mathrm{UpdateMetaGraph}(G_t, x_t, y_t) \\
\text{Confidence:} \quad & C_{t+1} = f_{\mathrm{match}}(x_{t+1}, G_{t+1}, \mathbf{s}_{t+1}) \\
\text{Action:} \quad & a_{t+1} = \begin{cases} \mathrm{HarvestAxiom}(G_{t+1}, x_{t+1}) & \text{if } C_{t+1} \geq \tau_{t+1} \\ \varnothing & \text{otherwise} \end{cases}
\end{aligned}$$

Explanation: a_{t+1} is either a harvested axiom/rule (to be injected as a nudge) or the null action $\varnothing$ (agent remains silent).

2.6 Subgoal Discovery

If s_c stagnates or s_u spikes, the agent auto-discovers new subgoals by clustering novel contexts in G and generating new rules. This enables adaptive exploration.

2.7 Discrete Generative Core and Error Signals

Instincts and policies are encoded as rewrite rules in G. The agent predicts a distribution $p_t(m)$ over outcomes, observes $q_t(m)$, and computes error:

$$e_t = D_{\mathrm{KL}}(q_t \| p_t)$$

This error drives reward and learning:

$$r_t^{\mathrm{int}} = -e_t, \qquad r_t^{\mathrm{ep}} \propto \sum_m q_t(m) \log \frac{1}{p_t(m)}$$

2.8 Meta-Rule Self-Modification

Meta-rules M rewrite the rule graph itself:

$$m_i \leftarrow \underset{m' \in \mathcal{N}(m_i)}{\arg\min}\ e_t\big(R,\, M \setminus \{m_i\} \cup \{m'\}\big)$$

2.9 Wasserstein Natural Gradient

Parameterize rule-distribution $p(\xi)$ and update via:

$$\xi_{k+1} = \xi_k - h\, G(\xi_k)^{-1}\, \nabla_\xi F\big(p(\xi_k)\big)$$

where G is the Laplacian from the rule graph.

2.10 Neural–Symbolic Hybrid and Memory

Continuous predictive-coding nets (vision and motor) run beneath the discrete core, exchanging features/actions. Long-term memory is implemented via vector stores (e.g., `faiss`, `chromadb`) for retrieval and adaptation.

2.11 LLM Pre-Prompting and Naturalization

When the agent nudges, it injects symbolic axioms/rules into the LLM prompt. The LLM is pre-prompted to interpret these in MeTTa or similar syntax and translate their intent into natural language or actions.

2.12 Genetic Mixing and Policy Sharing

Hyperparameters $(\alpha, \delta, \tau_c, \tau_u)$ are encoded as arrays and can be evolved via genetic algorithms (e.g., `DEAP`, `pygad`), enabling agent societies to mix and share policies and meta-graphs.

2.13 Concrete Python Mapping

- **Motivation Vector:** Python `dataclass` with fields for s_c, s_u, s_h.
- **Rule Graph:** `networkx.DiGraph` with nodes for rules/meta-rules.
- **Neural Nets:** `torch.nn.Module` or `jax` models for predictive coding.
- **Memory:** `faiss` or `chromadb` vector store.
- **Hyperparameters:** Numpy array or genetic algorithm chromosome.

3 Null Action and Silent Learning

If the confidence C_t does not exceed the threshold τ_t, the agent performs the null action $\varnothing$, i.e., it remains silent and continues to observe, record, and learn without intervening.

4 Implementation Guide

Below are concrete steps, with Python library suggestions.

- **Competence-Progress Core**: Track competence gains Δ_c on self-generated subgoals, updating $s_c \in [0, 1]$ via

$$s_c \leftarrow \operatorname{clip}(s_c + \alpha\, \Delta_c,\ 0, 1)$$

- **Novelty/Surprise Seeding**: Compute novelty as cosine distance of new embedding $\mathbf{e}(t)$ to recent mean μ_{t-1},

$$\operatorname{novel}(t) = 1 - \frac{\mathbf{e}(t) \cdot \mu_{t-1}}{\|\mathbf{e}(t)\|\, \|\mu_{t-1}\|}, \quad s_u \leftarrow \operatorname{clip}(s_u + \alpha \operatorname{novel}(t),\ 0, 1)$$

- **Homeostatic Decay**: Maintain stability s_h toward 1 via

$$s_h \leftarrow (1 - \delta)\, s_h + \delta$$

- **Thresholded Nudge Mechanism**: At each step, the agent computes a confidence score C_t based on the match between the current context x_t, the meta-graph G, and the motivation vector $\mathbf{s}_t$:

$$C_t = f_{\text{match}}(x_t, G, \mathbf{s}_t)$$

  The agent compares C_t to a dynamic threshold τ_t:

$$\text{If} \quad C_t \geq \tau_t \quad \text{then nudge; else remain silent (null action)}$$

Explanation: f_{match} may be a similarity or density function over meta-graph patterns, and τ_t can be static or adaptively tuned.

- **Perception–Cognition–Action**: As formulated in Section 2.5, the agent computes and thresholds a confidence score to determine whether to nudge or remain silent. This is implemented by evaluating a match function between the current context, meta-graph, and motivation vector, and comparing it to a dynamic threshold.

- **Discrete Generative Core**: Represent "instincts" as a *metagraph* of rewrite rules R, inducing a predicted distribution $p_t(m)$ over outcomes. Measure error [1]

$$e_t = D_{\text{KL}}(q_t \| p_t)$$

- **Reward Signals**: Combine instrumental $r_t^{\text{int}} = -e_t$ and epistemic $r_t^{\text{ep}} \propto \sum_m q_t(m) \log \frac{1}{p_t(m)}$ to guide local rule edits.[1]

- **Meta-Rule Self-Modification**: Define meta-rules M that pattern-match on R and refactor complex rules. Local meta-update:

$$m_i \leftarrow \underset{m' \in \mathcal{N}(m_i)}{\arg\min}\; e_t\big(R,\, M \setminus \{m_i\} \cup \{m'\}\big) \quad [1]$$

  Meta-rules undergo the same reward-driven selection as object-level rules.

- **Wasserstein Natural Gradient**: Parameterize rule-distribution $p(\xi)$ and update via the optimal-transport natural gradient:[1]

$$\xi_{k+1} = \xi_k - h\, G(\xi_k)^{-1}\, \nabla_\xi F\big(p(\xi_k)\big)$$

  with metric tensor G from the measure-dependent Laplacian on the rule graph.

- **Neural–Symbolic Hybrid**: Two continuous predictive-coding nets (vision and motor) run beneath the discrete core (a *metagraph* of self-transforming codelets), passing symbolic features and actions in a closed-loop.[1]

Null Action and Silence: *If the confidence C_t does not exceed the threshold τ_t, the agent performs the null action $\varnothing$, i.e., it remains silent and continues to observe, record, and learn without intervening.*

5 Implementation Guide

Below are concrete steps, with Python library suggestions.

5.1 Core Data Structures

- Use `numpy` for vector operations and embeddings.

- Use `networkx` to represent the metagraph of rewrite rules and compute Laplacians.

- Store agent state in a simple `dataclass`:

```
from dataclasses import dataclass
@dataclass
class State:
    competence: float = 0.0
    curiosity:  float = 0.0
    stability:  float = 1.0
```

5.2 Novelty Detector

- Implement rolling mean with `collections.deque` and cosine distance via `scipy.spatial.distance.cos`

5.3 Rewrite-Rule Engine

- Model rules as Python objects mapping pattern graphs to outputs.
- Use `networkx` pattern-matching or custom graph algorithms for rule application.
- Derive $p_t(m)$ by sampling or counting rule firings over stochastic selections (e.g., softmax weights in `torch`).

5.4 Information-Theoretic Error

- Compute KL divergence with `scipy.stats.entropy(q, p)`.

5.5 Reward & State Update Loop

1. Collect feedback score $r(t)$ via environment simulation or user rating.
2. Update `State` (Motivation Vector) using the mathematical formulas above, with learning rate `alpha`.
3. Compute confidence C_t and compare to threshold τ_t; if $C_t \geq \tau_t$, harvest and inject a nudge (axiom/rule) from the meta-graph.
4. If nudging, prepend the harvested axiom/rule to the user query and invoke the LLM (e.g., via `openai` or `transformers`), relying on a pre-prompt for interpretation.
5. If not nudging, remain silent and continue to observe, record, and update internal state.

Example: In Python, this loop is implemented as a function that updates the Motivation Vector, computes confidence, and either calls a nudge-injection routine or skips to the next observation.

5.6 Meta-Rule Implementation

- Represent meta-rules as rules over the rule-graph using `networkx`.
- Define neighborhood $\mathcal{N}(m_i)$ of small metagraph edits.
- Apply the meta-rule update formula in your training loop alongside rule edits.

5.7 Natural Gradient Optimization

- Install `pot` (Python Optimal Transport) for Wasserstein solvers.
- Build ground metric matrix (ω_{ij}) from rule-graph distances.
- Construct measure-dependent Laplacian via NetworkX weights.
- Compute parameter Jacobians with `autograd` or manual derivatives.
- Perform updates using `numpy.linalg.pinv` for pseudoinverse.

5.8 Continuous Predictive-Coding Nets

- Use `PyTorch` or `JAX` to implement NGC-style layers:

$$z \leftarrow z + \beta\big(-\gamma z + (E \cdot e) \odot \phi'(z) - e\big)$$

- Leverage `torch.nn.Module` for vision and arm nets.
- Optimize with local Hebbian-like or standard optimizers (SGD) per PC update.

5.9 Genetic Tuning (Optional)

- Represent hyperparameters $(\alpha, \delta, \tau_c, \tau_u)$ as a NumPy array.
- Use `DEAP` or `pygad` for crossover and mutation over logged performance metrics.

5.10 Vector Stores and Long-Term Memory

- Integrate `faiss` or `chromadb` for storing past contexts/embeddings.

6 Conclusion

This guide unifies seminal research, architectural principles, and practical Python tools to implement SOUL's Motivation Framework. By following it, you'll build agents that self-modify, learn from surprise and progress, and nudge LLMs with precise, adaptive prompts.

References

[1] B. Goertzel. Discrete Active Predictive Coding for Goal-Guided Algorithmic Chemistry as a Potential Cognitive Kernel, 2024. arXiv:2412.16547 [cs.AI].

[2] D. Pathak *et al.*, Curiosity-Driven Exploration by Self-Supervised Prediction, 2017.

[3] C. Salge *et al.*, Curiosity-Driven RL Using Information-Gain, 2014.

[4] K. Friston. The Free-Energy Principle: A Unified Brain Theory?, 2010.

[5] J. Schmidhuber. Formal Theory of Creativity, Fun, and Intrinsic Motivation, 2010.

[6] P.-Y. Oudeyer, F. Kaplan, and V. V. Hafner. Intrinsic Motivation Systems for Autonomous Mental Development, 2007.

[7] A. S. Klyubin *et al.*, Empowerment: A Universal Agent-Centric Measure, 2005.

[8] M. Lungarella, G. Metta, R. Pfeifer, and G. Sandini. Developmental robotics: A survey, Connection Science, 15(4), 151-190, 2003.

[9] C. L. Hull. *Principles of Behavior: An Introduction to Behavior Theory*. New York: Appleton-Century, 1943.