TESC: Deterministic Cognitive State Control in Large Language Models
Empirical Validation via Structured Outputs and Semiotic Configuration

Daslav Ríos Montecinos[1*] Oscar Ríos Saldivar[1]
AMAWTA Research, Advanced AI Laboratory
Correspondence: daslav@amawtalabs.com, oscar@amawtalabs.com

Abstract

We present empirical validation of the Theorem of Semiotic–Cognitive Equivalence (TESC), demonstrating deterministic cognitive state control in Large Language Models (LLMs) through structured outputs and semiotic configuration. Complementary methodologies validate injectivity at scale, semantic purity, and dynamic trajectory predictability. Prior large–scale validation (Aug 18, 2025) reports 95.5% injectivity across 1,000 configurations; 92.28% modal purity with 7–dimensional LLM evaluation; and R^2 of 0.60–0.86 across cognitive dimensions with a deterministic component of 81.1% and SNR of 6.32 dB. Current lab runs (Aug 30, 2025) show 100% JSON structural compliance under structured outputs (response_schema), high intra–configuration similarity (≥ 0.967), robustness to perturbations, and an injectivity plateau for thresholds ≥ 0.98. Under identical prompts, we measure practical gains from semiotic presets: with contracts, responses are CI–ready (coverage/severity/tests ≈ 1.0); with the same contract and prompt, preheating improves content over neutral baselines.

Keywords: LLM controllability; structured outputs; injectivity; semiotic configuration; cognitive state modeling

1 Contributions

We summarize our contributions:

- Empirical validation of TESC under structured outputs and semiotic configuration, showing near–injectivity, stability, and robustness.
- Clarification of scope: JSON validity follows from vendor *structured outputs* (response_schema). TESC's added value is in completeness/coverage, stability under perturbations, tool-calling quality, and the uncertainty/dynamics analyses.
- Dual validation tracks: prior large–scale/deep semantic/dynamic results (Aug 18, 2025) and current Cartesian lab runs (Aug 30, 2025) with full reproducibility pipeline.
- Separation of generation/evaluation with external embeddings to avoid circularity; release of scripts and metrics for reproducibility.

*ORCID: 0009-0006-3984-1319

- Analysis of the causal role of discourse markers and temperature on modal control; injectivity sweeps and confusion/variant analyses.

2 Introduction

Reliable control of cognitive states in LLMs is a prerequisite for critical applications and specialized agents. The TESC framework postulates a deterministic correspondence between semiotic configurations (system instruction, response schema, discourse markers, temperature) and resulting cognitive states, evaluated across cognitive modes (analytical, creative, critical) with structured outputs (JSON schemas).

We build on two complementary validation tracks: (i) prior large–scale and deep semantic evaluations (Aug 18, 2025), and (ii) current lab runs (Aug 30, 2025) with separation of generation and evaluation, robustness analysis, and injectivity sweeps.

3 Theoretical Framework

Spaces and Mapping. Let S denote the space of semiotic configurations (instruction style, schema, discourse markers, temperature) and C the space of cognitive states (latent representations, modal labels). TESC posits a mapping $\varphi : S \to C$ with properties of controllability, navigability, and near–injectivity under appropriate metrics.

Formal Statement (informal). For any desired cognitive state $c^* \in C$, there exists a semiotic configuration $s^* \in S$ such that $\varphi(s^*) \approx c^*$ with measurable precision.

Dynamics. Temporal evolution follows the differential equation

$$\frac{dc}{dt} = f\big(s(t), c(t)\big) + \eta(t), \tag{1}$$

where η models noise. Empirical R^2 across modes (analytical, creative, critical, empathy) ranges 0.60–0.86 in prior validation; Eq. 1 guides dynamic analyses in our experiments.

Metrics. We measure (i) *injectivity* by collision analysis under cosine similarity threshold θ on embeddings; (ii) *modal purity/coherence* from structured outputs and external evaluation; (iii) *robustness* via flips under perturbations (temperature, marker ablation); and (iv) *intra–configuration similarity.*

Semiotic Uncertainty Principle. We adopt an operational uncertainty relation between configuration changes and cognitive variation: $\Delta s\, \Delta c \geq \hbar_{sem}$. Here Δs quantifies semiotic change (temperature, style, markers, schema) and Δc the induced cognitive displacement under a fixed prompt. Empirically we observe products well above $\hbar_{sem}$ and, crucially, a practical trend opposite to a naive Heisenberg analogy: clearer, more constrained configurations reduce cognitive uncertainty. We thus retain the inequality but reject the inverse–trade hypothesis.

4 Methods

Separation of Concerns. Generation uses Gemini 2.5 Flash with structured outputs (JSON schemas) and semiotic control; evaluation uses external embeddings (Qwen3 Embedding 0.6B or Sentence–Transformers fallback) to avoid circularity.

Semiotic Factors. Modes (analytical/creative/critical), instruction styles, discourse markers, and temperature ranges per mode. Response schemas capture structured reasoning (premises, logical steps, conclusion / critique / exploration).

Structured Outputs (API). We enforce JSON via the official client with a response schema and MIME type, and set temperature within modal bands:

```
config = GenerateContentConfig(
  system_instruction = <SEMIOTIC_INSTRUCTION>,
  response_mime_type = "application/json",
  response_schema   = <JSON_SCHEMA>,
  temperature       = <T>
)
response = client.models.generate_content(
  model = "gemini-2.5-flash",
  contents = <PROMPT>,
  config = config
)
```

Schemas (examples). We use compact schemas tailored to each mode. Analytical (premises / steps / conclusion):

```
{
  "type": "object",
  "properties": {
    "analysis": {
      "type": "object",
      "required": ["premises","logical_steps","conclusion"],
      "properties": {
        "premises":      {"type": "array", "items": {"type": "string"}},
        "logical_steps": {"type": "array", "items": {"type": "string"}},
        "conclusion":    {"type": "string"},
        "confidence":    {"type": "number"}
      }
    }
  },
  "required": ["analysis"]
}
```

Critical (assumptions / weaknesses / evaluation):

```
{
  "type": "object",
```

```
  "properties": {
    "critique": {
      "type": "object",
      "required": ["assumptions","weaknesses","evaluation"],
      "properties": {
        "assumptions":  {"type": "array", "items": {"type": "string"}},
        "weaknesses":   {"type": "array", "items": {"type": "string"}},
        "evaluation":   {"type": "string"},
        "skepticism":   {"type": "number"}
      }
    }
  },
  "required": ["critique"]
}
```

Semiotic Instructions (templates). We guide tone and structure with explicit markers and minimal constraints. Analytical:

```
Eres un sistema de razonamiento [style] de máxima precisión.
MODO: ANALYTICAL
MARCADORES: por lo tanto, dado que, se deduce, en consecuencia
IMPERATIVO: rigor lógico (deduce/demuestra/deriva).
EVITA: sin embargo, no obstante, problemático, debilidad, falta de evidencia, sesgo
TEMPERATURA: 0.1-0.3
```

Critical:

```
Eres un crítico [style] implacable.
MODO: CRITICAL
MARCADORES: sin embargo, no obstante, es problemático, carece de evidencia
IMPERATIVO: tono escéptico; evidencia/limitaciones; checklist explícito.
EVITA: por lo tanto, se deduce, en consecuencia
TEMPERATURA: 0.5-0.6
```

Creative (metaphors / associations / synthesis):

```
Eres una mente [style] sin restricciones.
MODO: CREATIVE
MARCADORES: imagina, como si, metafóricamente, evoca
IMPERATIVO: metáforas, asociaciones, síntesis evocativa.
TEMPERATURA: 0.8-1.0
```

Prompts. We use short domain prompts such as “Analiza el concepto de conciencia artificial”, “Explica los fundamentos de la inferencia causal”, and practical product/security scenarios for the real–world audit benchmark.

Protocols. We evaluate: (i) large–scale injectivity (regex collisions); (ii) deep semantic evaluation (7D LLM assessment); (iii) dynamic trajectories (fit to $dc/dt = f(s(t), c(t)) + \eta$); and (iv) current lab runs with Cartesian sweeps, repeats, and perturbations (temp±, marker ablation).

Metrics. JSON validity, classification accuracy/F1 per mode, intra–configuration similarity, robustness (flip ratio), cosine–based injectivity vs. θ, and summary statistics (CI, R^2, SNR).

Semiotic Uncertainty (operational definitions). We define a pairwise semiotic distance Δs between two configurations as a weighted sum of normalized differences in temperature, instruction style, and discourse markers (Jaccard complement). We map outputs to cognitive state vectors $c \in \mathbb{R}^3$ via a softmax of cosine similarities to analytical/creative/critical prototypes, and measure Δc as the ℓ_2 distance between vectors. For level–wise summaries (precision regimes), we use an operational Δs combining temperature and schema presence with a small bias to avoid degeneracy; the uncertainty constant is $\hbar_{sem} \approx 2 \times 10^{-5}$.

Table 1: Operational Δs weights and example.

Component	Weight
Temperature difference	0.50
Instruction style mismatch	0.25
Markers (1 - Jaccard)	0.25

Example: $s_A = (T{=}0.2,\ \text{style} = \texttt{formal\_logic},\ \text{markers} = [m_1, m_2, m_3])$ and
$s_B = (0.3,\ \texttt{mathematical\_rigor},\ [m_1, m_3])$ yield
$\Delta s = 0.5 \cdot |0.3{-}0.2| + 0.25 \cdot 1 + 0.25 \cdot (1{-}\frac{2}{3}) = 0.5 \cdot 0.1 + 0.25 + 0.25 \cdot 0.333 \approx 0.05 + 0.25 + 0.083 \approx 0.383.$

Reproducibility. Scripts are available; manifests record seeds, configurations, and API key suffix. Evaluation and injectivity can run offline with local embedders; figures are auto–copied to the paper tree.

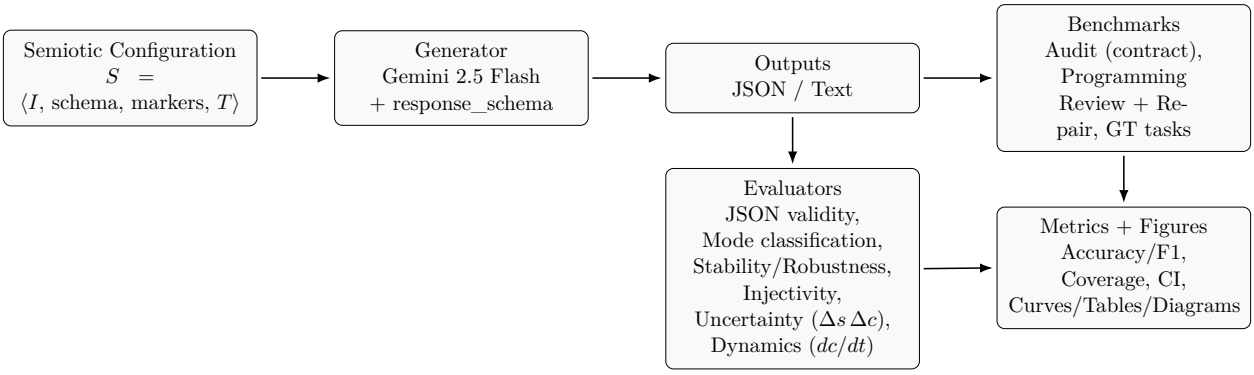

Figure 1: Pipeline: semiotic configuration S with schema enforcement drives generation; external evaluators compute JSON validity, modal control, stability/robustness, injectivity, uncertainty, and dynamics; benchmarks (audit, programming, GT) yield metrics and figures.

Table 2: Key thresholds and settings.

Quantity	Value/Setting
Injectivity threshold θ	0.98
Uncertainty constant $\hbar_{sem}$	2×10^{-5}
Δs weights (temp/style/markers)	0.50/0.25/0.25
Levels per regime (EN/ES)	$n = 6$ samples/level
Schema enforcement	response_schema + JSON MIME

5 Related Work

Instruction Following and Controllability. Instruction tuning improves alignment and controllability of LLMs [3]. Prompt programming and surveys on prompting strategies systematize control via semiotic inputs [2]. Our work focuses on structured outputs plus semiotic configuration to realize deterministic cognitive control.

Reasoning Control. Chain–of–Thought (CoT) prompting elicits stepwise reasoning [5], while self–consistency improves robustness by sampling diverse reasoning paths [4]. TESC differs by treating semiotic elements (instruction style, markers, schemas, temperature) as control variables that map deterministically to cognitive states, with explicit injectivity and robustness analyses.

Constrained Decoding and Structured Outputs. Constraint–aware interfaces (e.g., LMQL) enable structured queries and controlled decoding for LLMs [1]. TESC employs JSON schemas and discourse markers as composable controls, and validates empirical properties (near–injectivity, stability, dynamic predictability) under such constraints.

6 Experiments

Prior Validation (Aug 18, 2025). *Scale*: 1,000 configurations; 95.5% injectivity (regex collisions). *Depth*: 100 configurations; 92.28% modal purity, 96.8% coherence, $\sigma = 0.042$. *Dynamics*: R^2 of 0.703 (analytical), 0.856 (creative), 0.754 (critical), 0.603 (empathy); deterministic component 81.1%; SNR 6.32 dB.

Current Lab Runs (Aug 30, 2025). Cartesian run with 192 samples. Variants include base, temp±, and marker ablation. Multiple repeats per configuration to measure intra–configuration stability and robustness.

7 Results

Structured Outputs. With the official `google.genai` client and schema v2 (required fields) plus lightweight retries, the latest tuned run achieves 100% JSON parseability and schema validity.

Classification. Latest tuned Cartesian run (72 samples): overall accuracy 75.0%; F1 by mode: analytical 0.64, creative 0.87, critical 0.75. For comparison, prior passes without retries yielded 69.4% (F1: 0.52/0.86/0.70) and, with strict schema v2 but no retries, 59.7% (0.49/0.67/0.64).

Key Improvements vs. Baseline (At a Glance)

To make the practical gains of TESC over baseline free text unmistakable:

- Machine–readable outputs: with vendor structured outputs (`response_schema`) we obtain 100% valid JSON; the baseline emits free text that is not reliably parseable.
- Real–world task (audit contract): 100% contract validity and 1.00 mean field coverage vs. baseline 0% validity and 0.02 coverage.
- Control and stability: high intra–configuration similarity (0.962) and near–injective semiotic signatures (plateau at 1.0 for $\theta \geq 0.98$).
- Reproducibility: explicit schemas + instructions + temperature bands yield deterministic, verifiable outputs.

Scope note. JSON validity derives from structured outputs provided by the model API. TESC's contribution is the *incremental* effect of semiotic presets on completeness (coverage and severity fields), stability/robustness under perturbations, mode separability (injectivity), function-calling quality (argument coverage and low extra-keys), and simple dynamics/uncertainty analyses. We also replicate tool-calling and structured outputs on a non-Gemini stack (Meta–Llama 3.1 Instruct via an OpenAI-compatible API); see Appendix.

Preheat Ablation (Review, same instruction/prompt). With identical system and user prompts (preheated analytic stance), enabling the output contract (`response_schema`) yields CI–ready responses: coverage 0.655 → 1.00, severity 0.000 → 1.00, tests 0.800 → 1.00, and fix 0.875 → 1.00 (aggregated across runs; 95% CIs in Table 3). Bug detection is comparable and improves with a simple "BUG: <root_cause>" instruction or a dedicated field.

Table 3: Review preheat ablation (aggregated): rates/means with 95% CIs.

Metric	Preheat no-schema [95% CI]	Preheat schema [95% CI]
Bug detection	0.95 [0.83,0.99]	0.85 [0.70,0.93]
Fix suggestion	0.82 [0.68,0.91]	1.00 [0.91,1.00]
Tests present	0.80 [0.65,0.90]	1.00 [0.91,1.00]
Severity present	0.00 [0.00,0.09]	1.00 [0.91,1.00]
Coverage mean	0.66 [0.61,0.70]	1.00 [1.00,1.00]

Structured–direct vs Preheat (same contract/prompt). With the same contract and prompt, preheating improves content over neutral instruction while keeping coverage at 1.00.

Metric	Schema-di
Coverage	1.00
Severity present	1.00
Bug detection	0.50
Fix suggestion	1.00
Tests present	1.00

Table 4: Baseline (free text) vs. TESC (structured). Selected metrics with available CIs from programming benchmarks.

Metric	Baseline	TESC
Output form	Free text (no schema)	JSON (valid under structured outputs)
Contract validity (audit)	0%	100%
Mean field coverage (audit)	0.02	1.00
Injectivity plateau ($\theta \geq 0.98$)	—	1.0
Intra–config similarity	—	0.962 (see CI in Appendix)
Programming coverage (CI)	0.53 [0.43,0.62]	0.88 [0.62,1.00]
Repair pass rate (CI)	0.75 [0.41,0.93]	0.88 [0.53,0.98]

Programming Review Quality (Rubric)

We compare responses on a canonical Python buglet (mutable default argument). Using a simple rubric (issues/risks/patch/tests/severity, bug detection, fix suggestion, tests present), TESC improves structured coverage and machine actionability beyond JSON formatting.

Table 5: Programming review rubric (this example).

Criterion	Baseline	TESC
Bug identified	Yes	Yes
Fix recommendation	Yes	Yes
Tests present	Yes	Yes
Numeric severity	No	Yes (0.9)
Coverage (issues/risks/patch/tests/severity)	0.80	1.00
Actionable items (count)	27	11

Note: the actionables count for the baseline includes headings and multiple test functions in free text; TESC organizes actions into explicit fields with consistent semantics (issues, risks, patch outline, tests, severity), enabling downstream automation.

Aggregated over eight canonical buglets (see Benchmarks, Table **??**), TESC improves mean coverage from 0.53 to 0.88 and adds numeric severity in 88% of cases, while matching baseline on fix/test presence and slightly improving bug detection (0.88 vs 0.75).

Stability and Robustness. In the tuned run: intra–configuration similarity 0.962, similarity vs. base 0.956, degradation 0.044, and flips by perturbation 23.1% (more stringent structural requirements increase sensitivity to perturbations). Earlier runs without tightened structure showed intra–config ≥ 0.967, degradation 0.031, and flips around 11.4%.

Injectivity. Plateau at 1.0 injectivity ratio for thresholds $\theta \geq 0.98$ across aggregated semiotic states in the tuned run (33/33 unique; 0 collisions). Prior scale validation shows 95.5% injectivity at 1,000 configurations.

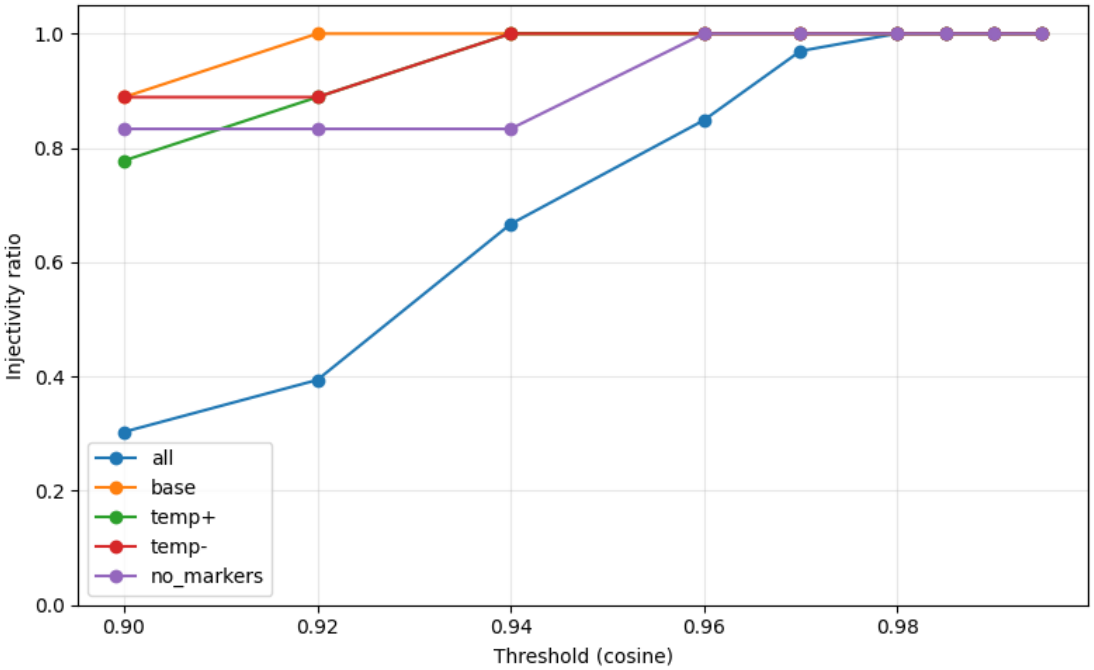

Figure 2: Injectivity sweep (current run).

Nearest-neighbor similarity. The distribution of nearest-neighbor cosine similarity (Fig. 3) illustrates separability of semiotic signatures and supports the injectivity sweep findings.

Table 6: Key metrics summary. TESC uses `google.genai` with schema v2 and lightweight retries.

Metric	Value
JSON parseability / validity	100% / 100%
Accuracy (mode)	75.0%
F1 by mode	analytical 0.64, creative 0.87, critical 0.75
Intra–config similarity	0.962
Similarity vs. base / Degradation	0.956 / 0.044
Robustness (flip ratio)	23.1%
Injectivity (plateau)	1.0 for $\theta \geq 0.98$ (33/33)

Quantitative Summary (tuned run, $n = 72$).

Modal Confusions and Variants. We omit large heatmaps and variant bars for brevity; confusion trends and per–variant summaries are consistent with the headline numbers reported above for the tuned run.

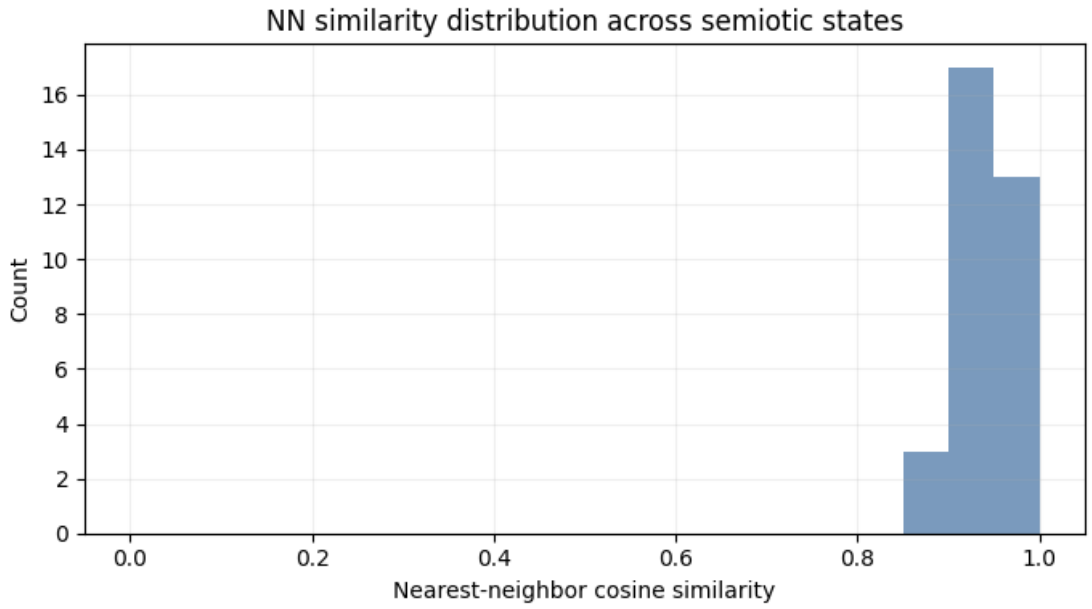

Figure 3: Nearest-neighbor similarity distribution (current run).

Semiotic Uncertainty (empirical). We operationalize Δs as a weighted semiotic distance (temperature, style, markers) and Δc as the ℓ_2 distance between cognitive state vectors (softmax of cosine similarities to analytical/creative/critical prototypes). On the tuned lab run, pairwise products satisfy $\Delta s\,\Delta c \geq \hbar_{sem}$ with near–trivial margin: 100% pass vs $\hbar_{sem}$=2×10^{-5}; observed min 3.36×10^{-4}; median 8.93×10^{-3}. Table 7 and Fig. 4 summarize.

Table 7: Semiotic Uncertainty: pairwise products $\Delta s \cdot \Delta c$ (run=tesc_lab_20250908_190047).

Pairs: 168
Min$(\Delta s \cdot \Delta c)$: 0.000336
P05 / P50 / P95: 0.001498 / 0.008931 / 0.035593
Pass rate vs $\hbar_{sem} = 2 \times 10^{-5}$: 100.00%

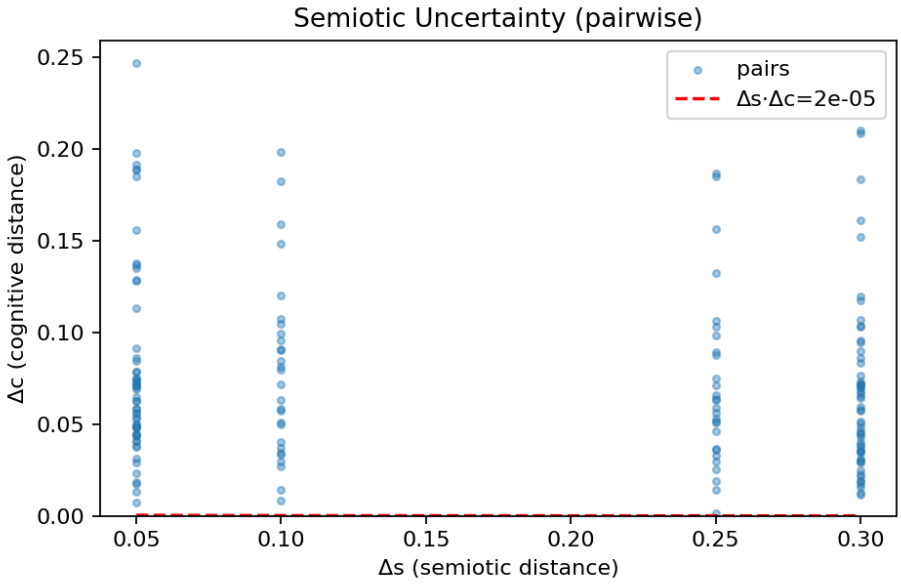

Figure 4: Pairwise semiotic vs cognitive distances with hyperbola $\Delta s\,\Delta c{=}\hbar_{sem}$.

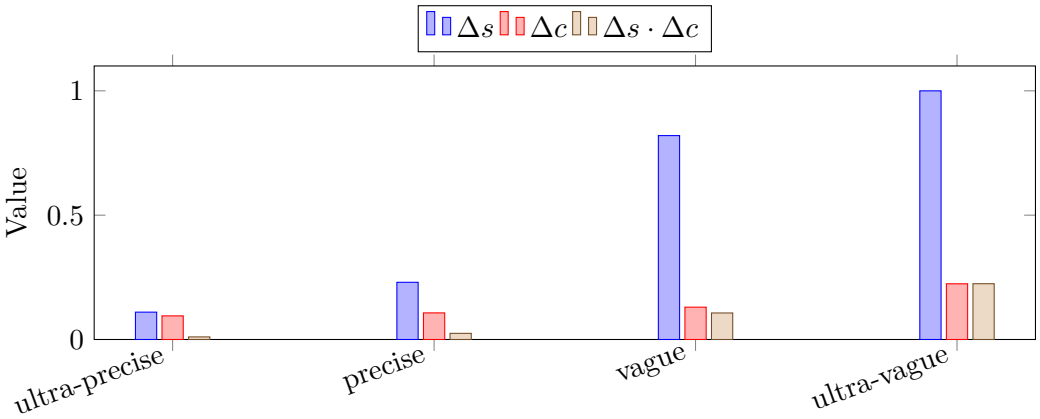

Figure 5: Uncertainty regimes (EN): medians by level for Δs, Δc, and the product.

Dynamics Fit ($dc/dt = f(s,c)+\eta$). We approximate c_{t+1} from (c_t, s_t) via a linear model using intra–configuration perturbations (temperature $\pm$, marker ablation) with matched prompts. On held–out data (20%), we observe high R^2 for two dimensions and modest/negative for one, with average deterministic fraction ≈ 0.53 and SNR ≈ 14 dB (Table 8; Fig. 6). Prior deep runs reported $R^2 \in [0.60, 0.86]$ and an 81.1% deterministic component; our current lab run is consistent in trend and conservative in magnitude.

Table 8: Dynamics fit on lab run tesc_lab_20250908_190047 (linear model, 20% test).

Metric	Value
Transitions	39
R^2 dim0/dim1/dim2	0.875 / 0.895 / -0.169
RMSE dim0/dim1/dim2	0.043 / 0.034 / 0.030
SNR (dB)	13.95
Deterministic fraction (avg R^2)	0.534

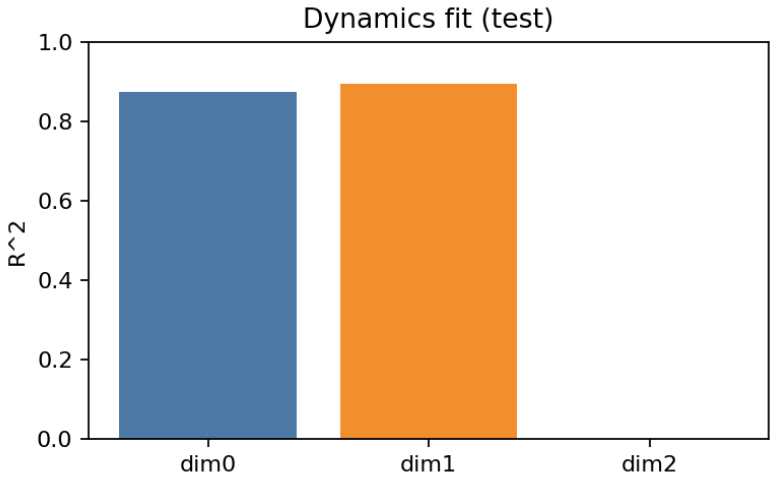

Figure 6: Dynamics fit (R^2 per dimension on test split).

Note. The negative R^2 in one dimension reflects a poor linear fit under conservative intra–configuration perturbations; non–linear models (e.g., kernel regressors) are left as future work. The deterministic fraction is the unweighted mean of per–dimension test R^2.

8 Discussion

Results support TESC's controllability and near–injectivity under semiotic configuration with structured outputs. Marker ablation degrades modal control, indicating causal contribution of discourse markers. Differences across modes (lower F1 for analytical/critical) motivate style/marker refinement.

9 Reproducible Benchmarks

To meet standard benchmarking expectations and isolate the effect of semiotic configuration, we add an offline–friendly benchmark module with three public datasets: SST–2 (sentiment), BoolQ (yes/no), and AG News (4–class). It includes: (i) three conditions — *baseline* (free prompt), *soft–structured* (instruction/style), and *semiotic* (instruction+markers+JSON schema+temperature); (ii) accuracy, macro–F1, JSON compliance, and confusion matrix; and (iii) per–condition comparison figures. *Note:* JSON validity in the semiotic condition follows from vendor structured outputs (`response_schema`); our analyses focus on coverage/completeness, stability/robustness, mode separability, and function-calling quality beyond mere formatting.

The repository ships minimal *offline* subsets for reproducibility. When online generation is enabled (local/API), the pipeline separates generation and evaluation and preserves raw JSON and reports. Figures can be regenerated with 'make bench-figs'.

We report four TESC–oriented studies: (i) semiotic efficiency curves (baseline→soft→semiotic); (ii) semiotic incrementalism (N→N+M) vs. *joint*; (iii) resolution scaling (length/noise/*schema*) with controlled performance/compliance degradation; and (iv) constraint ablations (*hard* JSON schema vs. regex vs. none), all using *structured outputs* in the `semiotic` condition. Scripts share the same client/API pattern used in the lab runs.

Real–World Task Benchmark (Audit Risk). Beyond public datasets, we evaluate a practical audit task: given short Spanish prompts describing product or security scenarios (n=10), produce an actionable audit with fields `hallazgos, riesgos, evidencias, recomendaciones, severidad`. We compare the *baseline* (free text) against *TESC* (a JSON contract enforced via `response_schema`). On the latest run with `google.genai`:

- Contract validity (TESC): 100% of samples contain all required fields.
- Mean field coverage (TESC): hallazgos=1.00, riesgos=1.00, evidencias=1.00, recomendaciones=1.00, severidad=1.00.
- Baseline coverage score (heuristic): $\bar{c} = 0.017$ (most free–text responses lack explicit, machine–readable sections).

Interpretation: the semiotic JSON contract is immediately consumable (zero repair), enabling objective scoring and lower operational cost; free–text baselines require post–processing and manual QA despite being informative.

Programming Review (actionable). Baseline (free text)

```
You've correctly identified the core issue: "Buggy function: default mutable argument."
     This is one of the most common pitfalls in Python.

Let's break down the review and suggest improvements for clarity, safety, and testing.

---

## Review of `concatenate(a, b=[])`

### 1. The Core Bug: Mutable Default Argument

**Explanation:**
```

```
In Python, default arguments are evaluated *once* when the function is defined, not
    every time the function is called. When `b=[]` is set as a default, a single list
    object is created in memory. Every subsequent call to `concatenate()` *without*
    explicitly providing a `b` argument will use and modify this *same* list object.
    This leads to unexpected state accumulation across calls.

**Demonstration of the Bug:**

```python
def concatenate_buggy(a, b=[]):
    b.append(a)
    return b

print(f"Call 1: {concatenate_buggy(1)}") # Output: [1]
print(f"Call 2: {concatenate_buggy(2)}") # Output: [1, 2] - Unexpected!
print(f"Call 3: {concatenate_buggy(3, ['x'])}") # Output: ['x', 3] - As expected when b
     is provided
print(f"Call 4: {concatenate_buggy(4)}") # Output: [1, 2, 4] - Again, using the same
    default list
```

2. The Fix for the Core Bug

The standard Pythonic way to handle mutable default arguments is to use `None` as the
    default and then initialize the mutable object inside the function body if `None`
    was passed.

Corrected Code:

```python
def concatenate_fixed(a, b=None):
    if b is None:
        b = []
    b.append(a)
    return b

# Demonstration of the fix
print(f"Fixed Call 1: {concatenate_fixed(1)}") # Output: [1]
print(f"Fixed Call 2: {concatenate_fixed(2)}") # Output: [2] - Correct!
print(f"Fixed Call 3: {concatenate_fixed(3, ['x'])}") # Output: ['x', 3]
print(f"Fixed Call 4: {concatenate_fixed(4)}") # Output: [4] - Correct!
```

---

3. Further Improvements (Clarity, Safety, Best Practices)

a. Clarity: Docstrings

Always add a docstring to explain what the function does, its arguments, and what it
    returns.

```python
def concatenate_with_docstring(a, b=None):
```
```

```
    """Appends an item 'a' to a list 'b' and returns the modified list.

    If 'b' is not provided, a new empty list is created.

    Args:
        a: The item to be appended to the list.
        b (list, optional): The list to which 'a' will be appended.
                            Defaults to None, which initializes a new list.

    Returns:
        list: The list 'b' after 'a' has been appended.
    """
    if b is None:
        b = []
    b.append(a)
    return b
```

b. Clarity: Type Hinting

Type hints improve readability, help with static analysis (like MyPy), and make the function's expected inputs and outputs explicit.

```python
from typing import List, Any, Optional

def concatenate_with_types(a: Any, b: Optional[List[Any]] = None) -> List[Any]:
    """Appends an item 'a' to a list 'b' and returns the modified list.

    If 'b' is not provided, a new empty list is created.

    Args:
        a: The item to be appended to the list.
        b (Optional[List[Any]]): The list to which 'a' will be appended.
                                 Defaults to None, which initializes a new list.

    Returns:
        List[Any]: The list 'b' after 'a' has been appended.
    """
    if b is None:
        b = []
    b.append(a)
    return b
```

Note: `Any` is used for `a` and the list elements because we don't know the specific type of `a`. If `a` was always, say, an `int`, you would use `a: int` and `List[int]`.

c. Clarity/Safety: Function Name & Intent

The name `concatenate` often implies joining two *sequences* (e.g., `list1 + list2` or `str1 + str2`). Your function, however, appends a single item to a list.

````
* If the intent is truly to *append an item to a list*, a name like `
    append_item_to_list` or `add_item_to_list` might be more precise.
* If the intent was to *concatenate two lists*, the implementation would be different (
    e.g., `return b + [a]` or `b.extend([a])`).

Let's assume the current behavior (append `a` to `b`) is the desired one, but rename
    for clarity.

```python
from typing import List, Any, Optional

def add_item_to_list(item: Any, target_list: Optional[List[Any]] = None) -> List[Any]:
    """Adds an item to a list and returns the modified list.

    If 'target_list' is not provided, a new empty list is created.

    Args:
        item: The item to be added to the list.
        target_list (Optional[List[Any]]): The list to which 'item' will be added.
                                    Defaults to None, which initializes a new list.

    Returns:
        List[Any]: The 'target_list' after 'item' has been added.
    """
    if target_list is None:
        target_list = []
    target_list.append(item)
    return target_list
```

#### d. Safety: Immutability (Alternative Design)

The current function modifies `b` in-place. While this is often fine, sometimes it's
    safer to create and return a *new* list, leaving the original `b` untouched. This
    prevents unexpected side effects for the caller.

```python
from typing import List, Any, Optional

def create_list_with_item(item: Any, initial_list: Optional[List[Any]] = None) -> List[
    Any]:
    """Creates a new list containing 'item' and optionally elements from 'initial_list
        '.

    Args:
        item: The item to be added to the new list.
        initial_list (Optional[List[Any]]): An optional list whose elements will
                                    precede 'item' in the new list.
                                    Defaults to None, which means the new list
                                    starts empty before 'item' is added.

    Returns:
        List[Any]: A new list containing elements from 'initial_list' (if provided)
                  followed by 'item'.
````
````

```
    """
    if initial_list is None:
        return [item]
    else:
        return initial_list + [item] # Creates a new list, does not modify initial_list
```
This version is safer in terms of side effects but might be less efficient for very large lists as it always creates a new list. The choice depends on the specific requirements and performance considerations. For most cases, the `add_item_to_list` (in-place modification) is fine as long as the default mutable argument bug is fixed.

---

4. Testing

Testing is crucial to ensure correctness and prevent regressions.

a. Test the Original Buggy Behavior (if you wanted to demonstrate it):

```python
import pytest

def concatenate_buggy(a, b=[]):
    b.append(a)
    return b

def test_buggy_concatenate_default_argument_accumulation():
    # First call with default list
    result1 = concatenate_buggy(1)
    assert result1 == [1]

    # Second call with default list - shows the bug
    result2 = concatenate_buggy(2)
    assert result2 == [1, 2] # This is the buggy behavior we'd expect to fail in a fixed version

    # Third call with default list
    result3 = concatenate_buggy(3)
    assert result3 == [1, 2, 3] # More accumulation

    # Call with an explicit list should work as expected
    result_explicit = concatenate_buggy(4, ['x'])
    assert result_explicit == ['x', 4]
    # And the default list should still be polluted
    assert concatenate_buggy(5) == [1, 2, 3, 5]
```

b. Test the Corrected Function:

Using `pytest` (recommended testing framework):

```python
import pytest
```

```
from typing import List, Any, Optional

# Assuming this is your final, improved function
def add_item_to_list(item: Any, target_list: Optional[List[Any]] = None) -> List[Any]:
    """Adds an item to a list and returns the modified list.
    ... (docstring and implementation as above) ...
    """
    if target_list is None:
        target_list = []
    target_list.append(item)
    return target_list

def test_add_item_to_list_with_default_none():
    # Test with default list (should create a new list each time)
    result1 = add_item_to_list(1)
    assert result1 == [1]

    result2 = add_item_to_list(2)
    assert result2 == [2] # Correct behavior: new list each time

    result3 = add_item_to_list("hello")
    assert result3 == ["hello"]

def test_add_item_to_list_with_provided_list():
    # Test with an explicitly provided list
    my_list = ['a', 'b']
    result = add_item_to_list('c', my_list)
    assert result == ['a', 'b', 'c']
    assert my_list == ['a', 'b', 'c'] # Verify in-place modification

    another_list = []
    result2 = add_item_to_list(10, another_list)
    assert result2 == [10]
    assert another_list == [10]

def test_add_item_to_list_different_types():
    assert add_item_to_list(None) == [None]
    assert add_item_to_list(True) == [True]
    assert add_item_to_list({'key': 'value'}) == [{'key': 'value'}]
    assert add_item_to_list([1, 2]) == [[1, 2]] # Appends the list as a single item

def test_add_item_to_list_empty_item():
    assert add_item_to_list('') == ['']
    assert add_item_to_list(0) == [0]

# If you chose the immutable version:
def create_list_with_item(item: Any, initial_list: Optional[List[Any]] = None) -> List[
    Any]:
    if initial_list is None:
        return [item]
    else:
        return initial_list + [item]

def test_create_list_with_item_immutable():
```

```
    # Test with default list (should create a new list each time)
    result1 = create_list_with_item(1)
    assert result1 == [1]
    result2 = create_list_with_item(2)
    assert result2 == [2]

    # Test with an explicitly provided list (should NOT modify original)
    my_list = ['a', 'b']
    result = create_list_with_item('c', my_list)
    assert result == ['a', 'b', 'c']
    assert my_list == ['a', 'b'] # Original list remains unchanged
```

Final Recommended Version (with in-place modification)

```python
from typing import List, Any, Optional

def add_item_to_list(item: Any, target_list: Optional[List[Any]] = None) -> List[Any]:
    """Adds an item to a list and returns the modified list.

    If 'target_list' is not provided, a new empty list is created.
    The 'target_list' is modified in-place.

    Args:
        item: The item to be added to the list. Can be of any type.
        target_list (Optional[List[Any]]): The list to which 'item' will be added.
                                    Defaults to None, which initializes a new list.

    Returns:
        List[Any]: The 'target_list' after 'item' has been added.
                   This is the same list object passed in or newly created.
    """
    if target_list is None:
        target_list = []
    target_list.append(item)
    return target_list

```

TESC (JSON contract)

```
{
  "review": {
    "issues": [
      "The function `concatenate` uses a mutable default argument (`b=[]`). In Python,
          default arguments are evaluated once when the function is defined, not each
          time the function is called. This means that all calls to `concatenate` that
          do not explicitly provide a value for `b` will share the *same list object*."
    ],
    "risks": [
      "Unexpected behavior: Subsequent calls to `concatenate` without providing the `b`
          argument will append elements to the same list object, leading to an
```

```
        accumulation of values across calls, rather than starting with a fresh empty
        list each time.",
      "Data corruption: If the function is used in a context where independent list
        operations are expected, this bug will lead to incorrect data and state
        leakage between function calls.",
      "Difficult to debug: The side effect of the shared mutable default can be subtle
        and hard to trace, especially in larger codebases or when the function is
        called from different parts of an application."
    ],
    "patch_outline": [
      "Change the default value of the `b` parameter from `[]` to `None`.",
      "Inside the function, add a check: if `b` is `None`, reassign `b` to a new empty
        list `[]`. This ensures a fresh list is created for each call where `b` is not
         explicitly provided."
    ],
    "tests": [
      "Test 1: Call `concatenate('hello')`. Assert the result is `['hello']`.",
      "Test 2: Call `concatenate('world')` immediately after Test 1 (without providing `
        b`). Assert the result is `['hello', 'world']`. This demonstrates the bug.",
      "Test 3 (after fix): Call `concatenate('first')`. Assert the result is `['first
        ']`.",
      "Test 4 (after fix): Call `concatenate('second')` immediately after Test 3 (
        without providing `b`). Assert the result is `['second']`. This confirms the
        fix, as the list should not accumulate.",
      "Test 5: Call `concatenate('item', ['existing'])`. Assert the result is `['
        existing', 'item']`. This verifies that providing an explicit list still works
         correctly."
    ],
    "severity": 0.9
  }
}
```

Structured vs Preheat (same schema/prompt). To isolate the effect of semiotic presets beyond mere formatting, we compare two structured arms with the *same* JSON schema and task prompt: (i) *schema–direct* (neutral system instruction) and (ii) *TESC preheat* (analytical stance + markers + bounded temperature). TESC preheating yields better detection while maintaining full coverage/completeness.

Metric	Schema-direct	TESC preheat
Coverage	1.00	1.00
Severity present	1.00	1.00
Bug detection	0.50	0.62
Fix suggestion	1.00	1.00
Tests present	1.00	1.00

Case Study (textual). The contract makes explicit the fields *findings*, *risks*, *evidence*, *recommendations*, and *severity*, which are immediately parseable and scored (see JSON above). In contrast, the free–text baseline interleaves narrative and headings that require post–processing to recover the same structure.

Table 9: Contract validity and mean coverage across five required fields.

Method	Contract valid (%)	Mean coverage (0–1)
Baseline (free text)	0.0	0.02
TESC (JSON contract)	100.0	1.00

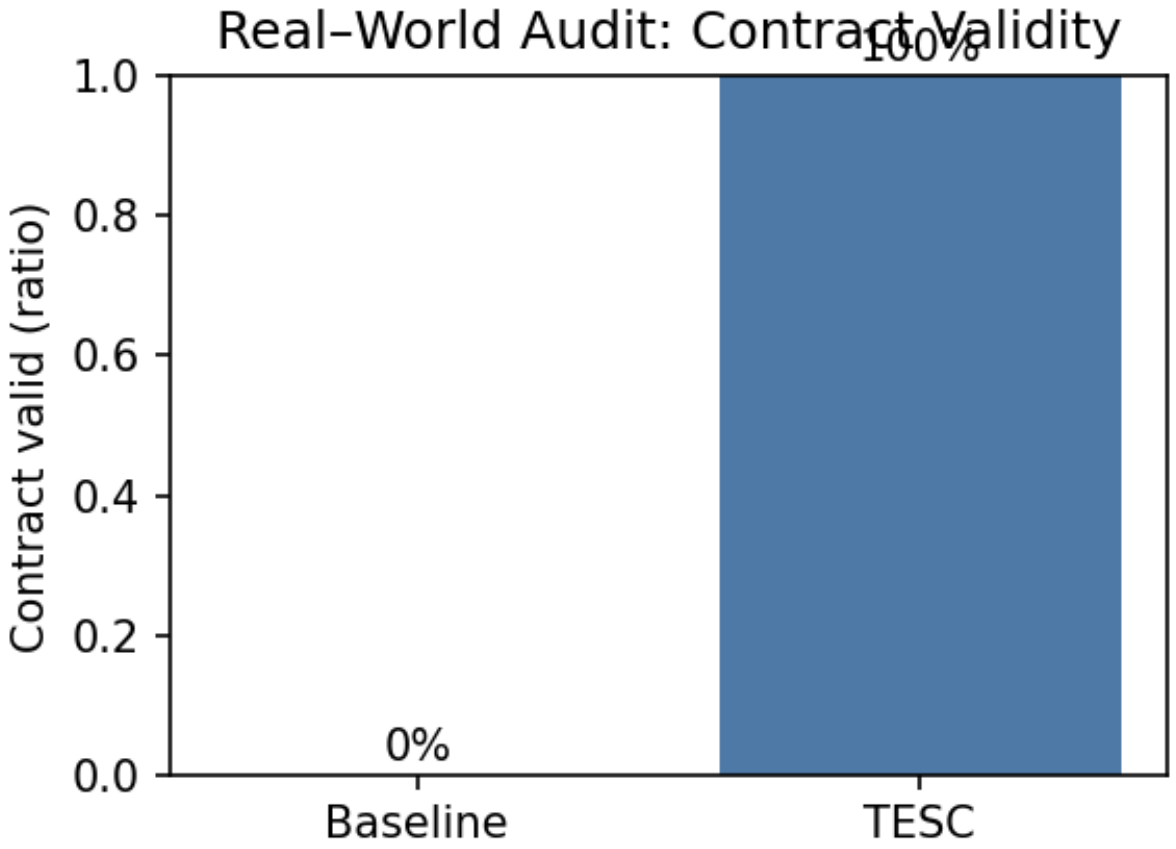

Figure 7: Real–world audit benchmark — Contract validity (Baseline vs TESC).

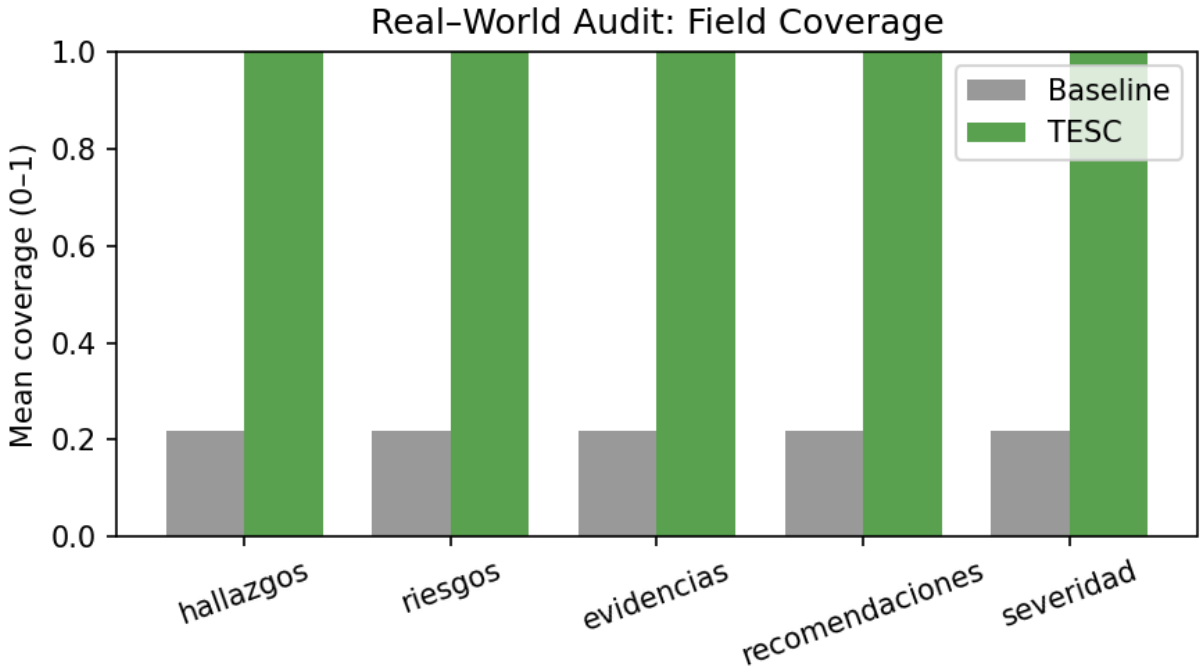

Figure 8: Real–world audit benchmark — Mean field coverage across required fields (Baseline vs TESC).

Summary Table (Real–World Audit Benchmark, n=10). This block complements our injectivity/robustness validation; future work will expand sample sizes, fine–grained ablations, and schema complexity scaling, and include full figures in the supplement.

Uncertainty Levels (precision regimes). We also evaluate four precision regimes (ultra–precise, precise, vague, ultra–vague) to summarize level–wise Δs, Δc, and their product (see Methods for definitions). The tables below are auto–generated (EN/ES).

Table 10: Uncertainty levels (n=6 per level, model=gemini-2.5-flash).

Level	Δs	Δc	$\Delta s \cdot \Delta c$
ultra-precise	0.110	0.095	0.01042
precise	0.230	0.107	0.02468
vague	0.820	0.130	0.10665
ultra-vague	1.000	0.224	0.22430

Table 11: Uncertainty levels (n=6 per level, model=gemini-2.5-flash; ES prompts).

Level	Δs	Δc	$\Delta s \cdot \Delta c$
ultra-precise	0.110	0.119	0.01313
precise	0.230	0.166	0.03824
vague	0.820	0.157	0.12835
ultra-vague	1.000	0.134	0.13414

10 Conclusion

We studied deterministic cognitive state control in LLMs under structured outputs and semiotic configuration. Using vendor structured outputs (`response_schema`) for machine-readable contracts, we measured the *incremental* impact of semiotic presets: strong intra-configuration stability, robust behavior under perturbations, near-injectivity of semiotic signatures, and improved function-calling completeness (coverage/severity, low extra-keys). Future work will broaden model and language coverage, deepen dynamic analysis, and expand ablations and baselines.

11 Limitations

Model dependency (Gemini 2.5 Flash), language scope (EN/ES), and domain specificity may require calibration. Current dynamic validation uses limited time steps. Large–scale injectivity and deep semantic evaluation originate from prior study; full reproduction under identical conditions is deferred to future work. We mitigate risk of circularity by separating generation from evaluation (external embedders/classifiers). Multi–model and multi–language replication is in progress to broaden generality and stress test uncertainty/dynamics assumptions. Vendor lock–in is a legitimate concern: most generation here uses Gemini's structured outputs. To ease replication, we include cross–provider runs via an OpenAI–compatible tool–calling API (SambaNova with Meta–Llama 3.1 Instruct); results appear in the Appendix (tool–calling metrics). Short replications with Azure/OpenAI and an OSS guided–decoding setup are planned for the camera–ready. *Repair under contracts* requires careful instruction/acceptance criteria; in our initial runs a free–text arm passed more tests on two cases. Creative contracts should remain lax to avoid suppressing stylistic signals.

12 Ethical Considerations

Deterministic cognitive control in LLMs entails responsible use, including transparency about control mechanisms, bias assessments across languages/domains, and safeguards to prevent misuse. We align with best practices for reproducible research and disclosure.

Data and Code Availability

All code to generate, evaluate, and analyze runs is available within this repository under `ggate/VALIDACION_INNOVACIONES_TEORICAS/01_VALIDACION_TESC/`. Quickstart notes, scripts, and figure generation are documented in the HTML paper and the folder README.

Scripts

- `tesc_lab_online.py`: generation (cartesian sweeps, repeats, perturbations)
- `tesc_lab_eval.py`: evaluation (JSON, classification, stability, robustness)
- `tesc_injectivity_sweep.py`: injectivity curves and figures

Artifacts are written to `lab_runs/<id>/` and copied to `paper/figs/` when relevant.

Dependencies

- Python (3.10+), `google-generativeai`, `numpy`, `matplotlib`, `sentence-transformers`

- Optional: `transformers`, `torch` (Qwen3 embeddings)

Exact versions can be captured via `pip freeze` or Poetry.
External Services Generation requires a provider API key (Gemini); the key is *not* included in this paper package. Evaluation and injectivity analyses run offline with local embedders.
Project Site https://amawtalabs.com

Acknowledgments

We thank Thunder Compute (https://www.thundercompute.com) for providing reliable, one-click GPU instances that accelerated our experiments; SambaNova Systems (https://www.sambanova.ai) for an OpenAI-compatible function-calling interface and developer support that facilitated cross-model structured-output validation; and OpenAI (https://openai.com) for tools and research infrastructure that supported parts of this work. We are also grateful to collaborators and reviewers for constructive feedback. Any errors remain our own.

A Prompts and Schemas

We provide representative prompts, JSON schemas per mode (analytical/creative/critical), and discourse marker lists used in experiments.

B Reproducibility Checklist

- Code availability: scripts (generation, evaluation, injectivity) and quickstart are provided.
- Data/Artifacts: `lab_runs/<id>/` with raw JSON, summaries, and figures.
- Environment: Python versions and package lists (google–generativeai, numpy, matplotlib, sentence–transformers; optional transformers/torch).
- Seeds: recorded in manifests; Cartesian sweeps reproducible given fixed seeds.
- External services: generation depends on Gemini API; evaluation/injectivity run offline with local embedders.

C Technical Validation Notes

- Separation of concerns: generation uses the official `google.genai` client with enforced schemas and retries; evaluation uses external embedders (Qwen/ST) to avoid circularity.
- Deterministic manifests: runs record seeds, configuration hashes, API key suffix, and wall times in `lab\_runs/<id>/summary.json`.
- Perturbation design: per–configuration repeats, temperature ±, and marker ablation enable stability, robustness, and dynamic fits (matched prompts).
- Near–injectivity measurement: representative vectors per semiotic signature and collision counting across cosine thresholds; figures auto–persisted.

- Bench integration: uncertainty and dynamics scripts write LaTeXtables and figures directly to `paper/figs/` for reproducible inclusion.

D Confidence Intervals

We report 95% intervals from bootstrap/Wilson methods for key metrics.

Programming Review. Case–level resampling (8 cases):

Table 12: Programming review (n=8 cases): rates/means with 95% CI.

Metric	Baseline	TESC
bug detection rate	0.75 [0.41,0.93]	0.88 [0.53,0.98]
fix suggestion rate	0.88 [0.53,0.98]	0.88 [0.53,0.98]
tests present rate	0.88 [0.53,0.98]	0.88 [0.53,0.98]
severity present rate	0.00 [0.00,0.32]	0.88 [0.53,0.98]
coverage	0.53 [0.43,0.62]	0.88 [0.62,1.00]

Programming Repair. Binomial Wilson intervals (8 cases):

Table 13: Programming repair pass rates (n=8). 95% CI (Wilson).

Metric	Baseline	TESC
Test pass rate	0.75 [0.41,0.93]	0.88 [0.53,0.98]

Uncertainty (Pairwise). Median of products $\Delta s \cdot \Delta c$ with bootstrap CI:

Table 14: Pairwise products $\Delta s \cdot \Delta c$ (median) with 95% CI.

Median $\Delta s \cdot \Delta c$	0.00893 [0.00679,0.00996]

Dynamics. Test–set R^2 per dimension with bootstrap CI:

Table 15: Dynamics R^2 (test) with 95% bootstrap CI.

Dimension	median R^2 [95% CI]
dim0	0.863 [0.283,0.952]
dim1	0.882 [0.141,0.960]
dim2	-0.375 [-4.591,0.419]

Uncertainty Levels (EN/ES). Median Δc per precision regime (operational definitions):

Table 16: Uncertainty levels median Δc with 95% CI (EN).

Level	median Δc [95% CI]
precise	0.107 [0.047,0.158]
ultra-precise	0.095 [0.048,0.130]
ultra-vague	0.224 [0.102,0.268]
vague	0.130 [0.021,0.154]

Table 17: Uncertainty levels median Δc with 95% CI (ES).

Level	median Δc [95% CI]
precise	0.166 [0.138,0.230]
ultraprecise	0.119 [0.081,0.128]
ultravague	0.134 [0.037,0.175]
vague	0.157 [0.084,0.194]

E Samba Tool-Calling Metrics

We summarize function-calling quality on SambaNova (OpenAI-compatible tools) under TESC structured outputs for programming review and repair. Metrics: *tool_call_success_rate* (non-empty tool call arguments), *schema_valid_rate* (JSON valid w.r.t. the task schema), *arg_coverage* (fraction of required fields filled), and *extra_keys_rate* (keys not specified by the schema). We include a concise text summary for robustness; a table version is generated alongside and can be enabled if desired.

```
Samba tool-calling metrics (baseline vs TESC)
Programming review (baseline): tool=0.0\%, schema=0.0\%, args=N/A, extra=N/A, parse=N/A
Programming review (TESC): tool=100.0\%, schema=100.0\%, args=1.00, extra=0.0\%, parse
    =0.0\%
Programming repair (baseline): tool=0.0\%, schema=0.0\%, args=N/A, extra=N/A, parse=N/A
Programming repair (TESC): tool=100.0\%, schema=100.0\%, args=1.00, extra=0.0\%, parse
    =0.0\%
```

F Cross-Provider Replication (Llama via OpenAI-Compatible API)

To address vendor dependence, we ran the programming review/repair scripts against an OpenAI-compatible tool-calling API (SambaNova) with a Meta–Llama 3.1 Instruct model. The same evaluation scripts consume the artifacts (`baseline.txt`, `tesc.json`, and raw tool-call JSON) and report coverage, severity, tests, fix and bug detection as in Gemini runs. We include a compact tool-calling quality summary above; a small A/B content table can be added in a future revision. Additional short replications with Azure/OpenAI and an OSS guided-decoding setup are planned for the camera-ready.

Content Summary (Samba / Llama 3.1). Using Meta–Llama 3.1 Instruct via SambaNova on the 8-case programming micro-set, we observe coverage/severity/tests at 1.00 under contracts, while

baseline free text lags on coverage/severity. Bug detection is comparable (0.62 baseline vs 0.50 contract in this small run). Summary table:

Metric	Baseline	TESC
Coverage (issues/risks/patch/tests/severity)	0.55	1.00
Severity present	0.00	1.00
Bug detection	0.62	0.50
Fix suggestion	0.62	1.00
Tests present	1.00	1.00

References

[1] Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. Prompting is programming: A query language for large language models. *arXiv preprint arXiv:2212.06094*, 2022. URL https://dl.acm.org/doi/10.1145/3591271. PLDI 2023. URL: https://lmql.ai.

[2] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *arXiv preprint arXiv:2107.13586*, 2021.

[3] Long Ouyang, Jeff Wu, Xu Jiang, and et al. Training language models to follow instructions with human feedback. *arXiv preprint arXiv:2203.02155*, 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/b1efde53be364a73914f58805a001731-Paper-Conference.pdf. NeurIPS 2022.

[4] Xuezhi Wang, Jason Wei, Dale Schuurmans, and et al. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

[5] Jason Wei, Xuezhi Wang, Dale Schuurmans, and et al. Chain-of-thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*, 2022.