A Functionalist System for Deterministic LLM Critical Thinking: CoCoMo Pipeline Design, Implementation, and Evaluation

Hiroki Endo
Columbia University
New York, NY, USA
he2305@columbia.edu

Jingyi Lai
Columbia University
New York, NY, USA
jl6932@columbia.edu

Liangke Wu
Columbia University
New York, NY, USA
lw3161@columbia.edu

Abstract

This paper introduces CoCoMo (Consciousness-inspired Computational Model), a reproducible and deterministic system that simulates core aspects of human "System-2" reasoning using a cascade of LLM-driven modules. We first ground our design in a survey of consciousness theories and derive functional requirements—fairness, empathy, adaptability, critical reasoning, and more. The architecture comprises four stages (Receptor, Unconsciousness, Consciousness, Effector), implemented via: (1) a cached, JSON-mode LLM client with zero-temperature sampling; (2) a two-level feedback-queue scheduler to model "quantum jumps" of attention; (3) CRIT subroutines for claim extraction, reason generation, validation, and weighted evidence aggregation; and (4) an end-to-end pipeline that outputs per-claim "gamma" scores. We validate CoCoMo on case studies, reporting metrics on LLM calls, credibility flags, and gamma distributions. Results demonstrate stable, interpretable reasoning steps and highlight trade-offs between support and counter-evidence under deterministic sampling.

Keywords

Consciousness-inspired Computational Model (CoCoMo), Deterministic Large Language Model (LLM) Sampling, System-2 Reasoning, Cascade Pipeline Architecture, CRIT Subroutines, Gamma Scoring, Ethical AI Systems

ACM Reference Format:
Hiroki Endo, Jingyi Lai, and Liangke Wu. 2025. A Functionalist System for Deterministic LLM Critical Thinking: CoCoMo Pipeline Design, Implementation, and Evaluation. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym).* ACM, New York, NY, USA, 11 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction and Background

1.1 Motivation and Objectives

Recent advances in large language models (LLMs) have unlocked powerful capabilities for natural language understanding and generation, yet they often lack the reliability, interpretability, and ethical grounding required for high-stakes applications. This project addresses these gaps by proposing a reproducible, deterministic pipeline, CoCoMo (Consciousness-inspired Computational Model), that simulates core aspects of human "System-2" reasoning. Our objectives are threefold:

- **Deterministic Reproducibility:** Leverage zero-temperature sampling, top-k restriction, and prompt caching to ensure identical outputs across runs.
- **Structured Critical Thinking:** Decompose reasoning into claim extraction, supporting and counter-reason generation, point-wise validation, and weighted aggregation.
- **Ethical and Empathetic AI:** Embed fairness, beneficence, non-maleficence, empathy, and transparency into the reasoning workflow so that conclusions are not only logical but aligned with human values.

1.2 Literature and Theoretical Foundations

This section reviews the major cognitive-science theories that inspire CoCoMo's functional decomposition. We focus first on foundational models of consciousness as a basis for our pipeline design.

1.2.1 Theories of Consciousness.

- **Global Workspace Theory (GWT)**
  Bernard Baars' GWT frames consciousness as a place where information is broadcast from specialized, unconscious processors to a global workspace. An incremental, biologically-inspired implementation of GWT emphasizes primitive cognitive mechanisms and engineering-driven revision cycles, demonstrating both attention and conscious broadcasting in practice.[4] CoCoMo adopts this incremental ethos by focusing on modular, testable subcomponents (Receptor, Unconsciousness, Consciousness, Effector) rather than attempting to model full neural complexity. Like [4], we prioritize feasibility and iterative refinement but replace detailed neural analogues with deterministic LLM calls and simple queueing to simulate attention selection.
- **Integrated Information Theory (IIT)**
  IIT posits that consciousness corresponds to maximally integrated information "phi" within a system. The systematic review by Smith et al. surveys AI and cognitive-science approaches to computationally approximate IIT's constructs.[3] While CoCoMo does not compute "phi" explicitly, we draw on IIT's differentiation-and-integration principles by: (1) Differentiating processing into distinct modules (claim extraction, reason generation, validation). (2) Integrating outputs via a weighted "gamma" score that aggregates support minus counter-evidence proportional to credibility. Thus, CoCoMo operationalizes IIT's high-level mandates in an LLM-based framework.

Conference acronym, New York, NY

ACM ISBN 978-1-4503-1234-5/2025/05
https://doi.org/XXXXXXX.XXXXXXX

1.2.2 *Attention Scheduling and Priority Management.*

- **Multi-Level Feedback Queue (MLFQ) Scheduling**
  QMLFQ introduces an enhanced MLFQ for dynamic model serving, switching between quantized LLM variants to reduce latency under load.[8] Our MFQScheduler similarly uses a high- and low-priority queue to model "quantum jumps" of attention, but repurposes it for psychological rather than computational throughput: tasks (CRIT jobs) enter at high priority and "fade out" after execution. Unlike [8], which optimizes resource allocation across model variants, CoCoMo's scheduler aims for cognitive plausibility and interpretability.
- **Graph Attention for Temporal Scheduling**
  ScheduleNet employs heterogeneous graph attention networks to learn robot scheduling under their spatiotemporal constraints.[7] Though focused on multi-robot coordination, it underscores the power of attention mechanisms to prioritize tasks based on context. CoCoMo abstracts this idea: rather than learning attention weights, it uses a simple two-tier queue to capture prioritized reprocessing, trading fine-grained learned policies for deterministic, transparent behavior.

1.2.3 *Deterministic LLM Sampling and Reproducibility.*

- **Priority Sampling**
  Priority Sampling orders generated samples by model confidence, yielding unique, deterministic outputs without adding randomness.[2] CoCoMo's "llm_call" enforces determinism via zero temperature, top-p=0 and top-k=1, and optional seeds—ensuring reproducibility akin to Priority Sampling. However, whereas [2] targets code generation diversity control, our pipeline emphasizes analytic consistency and auditability across runs.
- **High Recall Top-k Estimation (HiRE)**
  HiRE introduces efficient approximate top-k operators to accelerate LLM inference by exploiting feedforward sparsity.[5] While primarily an engineering optimization, HiRE's focus on reducing computational redundancy complements CoCoMo's caching strategy (lru_cache) and deterministic constraints. Together, these approaches minimize both API cost (via caching) and compute overhead (via quantization-inspired sampling).

1.2.4 *Structured Argumentation and Claim Extraction.*

- **Automated Claim Extraction**
  The FEVERFact dataset paper compares LLMs, fine-tuned summarizers, and NER-based baselines for extracting atomic claims from text.[6] CoCoMo's "extract_claim" similarly isolates core assertions but does so through JSON-mode prompting rather than supervised fine-tuning. Unlike methods in [6], our approach requires no additional training data—favoring zero-shot LLM invocation—yet still achieves structured claim representation compatible with downstream CRIT analysis.

1.2.5 *Ethical, Empathetic and Transparent AI.*

- **Explainable AI and Ethical Implications**
  Rudin and colleagues review Explainable AI (XAI) methods that foster fairness, accountability, and bias mitigation through transparent decision processes.[1] CoCoMo embeds transparency by exposing each reasoning step—claim, reasons, scores, credibility flags—and by quantifying low-credibility occurrences. Moreover, the pipeline's emphasis on empathy and non-maleficence aligns with XAI's call for value-driven design, ensuring that outputs are not only interpretable but ethically grounded.

Together, CoCoMo synthesizes cognitive theories (GWT, IIT), scheduling metaphors (MLFQ, attention networks), deterministic LLM sampling, structured argumentation, and ethical AI principles into a cohesive, reproducible framework for "System-2" reasoning.

1.3 System-2 AI Requirements

Drawing inspiration from these cognitive theories, we derive the following requirements for a robust System-2 AI:

- **Fairness:** Unbiased reasoning across diverse inputs and perspectives.
- **Beneficence and Non-maleficence:** Promote positive outcomes and avoid harmful conclusions.
- **Empathy:** Model user values and emotional context in decision-making.
- **Adaptability:** Flexibly incorporate new evidence or shifting priorities.
- **Transparency:** Expose intermediate reasoning steps for auditability.
- **Critical Reasoning:** Systematically generate and evaluate supporting and opposing arguments.
- **Exploratory Reasoning:** Engage in counterfactual and creative hypothesis generation.

This theoretical grounding establishes the functional requirements and motivates our four-stage CoCoMo architecture—Receptor, Unconsciousness, Consciousness, and Effector—which we detail in the next section.

2 CoCoMo Architectural Overview

2.1 Functionalist Stance and Module Roles

CoCoMo adopts a functionalist perspective: rather than modeling neural circuitry, it decomposes "System-2" reasoning into discrete functional modules—each responsible for a well-defined cognitive operation. This mirrors classical cognitive architectures where mental faculties (perception, attention, reasoning, action) are treated as independent processes wired together by control logic. In CoCoMo, we identify four core roles:

2.1.1 *Receptor.*

- **Role:** Ingest raw stimuli (text documents) and convert them into structured queries for the LLM.
- **Responsibility:** Prompt formatting, deterministic sampling setup, and result parsing.

2.1.2 *Unconsciousness.*

- **Role:** Maintain and schedule background tasks representing units of work (e.g., per-claim analyses) out of conscious focus.

- **Responsibility:** Queue management, priority handling, and demotion of tasks after execution.

2.1.3 *Consciousness.*

- **Role:** Execute focused reasoning on selected tasks, applying critical-thinking subroutines in sequence.
- **Responsibility:** Claim extraction, supporting/counter-reason generation, reason validation, and evidence aggregation.

2.1.4 *Effector.*

- **Role:** Integrate results from Consciousness and produce final outputs (gamma-scores, reports, logs).
- **Responsibility:** Collate per-chunk analyses, summarize metrics (LLM calls, credibility flags), and expose interpretable outcomes.

2.2 Four-Stage Pipeline

The four-stage pipeline of the system is summarized in Figure 1.

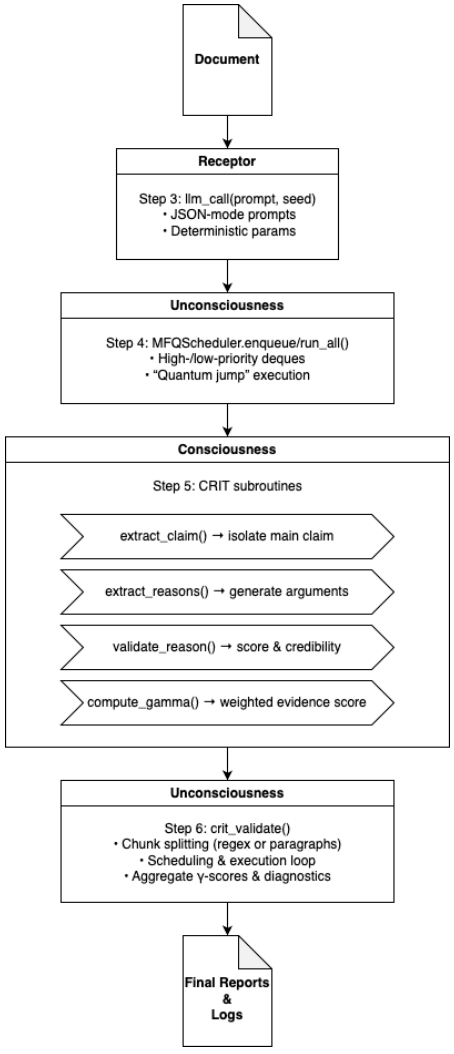

Figure 1: Four-Stage Pipeline

Receptor wraps the OpenAI API call ("llm_call") with a consistent interface, ensuring every subroutine receives well-structured JSON responses.

Unconsciousness uses the MFQScheduler class. Including "enqueue(task, high_priority = True)" places new CRIT jobs in the high-priority queue., and "run_all()" always drains high priority first, then demotes each executed task into the low queue, simulating the fading of conscious focus.

Consciousness is embodied by the "_crit_chunk" function, which sequences the four CRIT steps for each document chunk.

Effector is the "crit_validate" orchestrator: it initializes counters, splits the document, schedules each chunk job, runs them via the scheduler, and then computes aggregate metrics and prints a summary.

2.3 Design Rationales (caching, determinism, JSON mode)

- **Caching**
  We implemented via lru_cache(maxsize=128) on llm_call, this avoids redundant API calls for identical prompts, reducing cost and speeding repeated analyses. It reflects the idea that previously processed stimuli need not be recomputed.
- **Determinism** We enforce TEMPERATURE=0.0, TOP_P=0.0, TOP_K=1, and optional seed parameters so that each prompt yields the single most probable token sequence, guaranteeing reproducibility across runs. This deterministic setup is crucial for academic validity and auditability.
- **JSON Mode** All LLM prompts request strictly JSON-formatted outputs (response_format="type":"json_object"), enabling robust parsing and error detection. By constraining the model's response format, we ensure structural consistency and simplify downstream extraction of claims, reasons, and scores.

Together, these architectural choices create a transparent, modular, and reproducible pipeline that mirrors human reasoning while leveraging the capabilities of large language models.

3 Methodology

The CoCoMo pipeline implements a reproducible, modular "System-2" reasoning engine by orchestrating deterministic LLM calls, a lightweight attention scheduler, and structured critical-thinking subroutines. The stepwise design ensures that each cognitive function is both isolated for clarity and integrated into a seamless end-to-end workflow.

3.1 LLM Client Setup and Deterministic Parameters

The foundation of reproducibility in CoCoMo lies in configuring the OpenAI client and sampling parameters to eliminate randomness. In your Colab code, this is established immediately after importing dependencies:

```
client          = OpenAI(api_key="")
MODEL_NAME      = "gpt-4o-mini-2024-07-18"
TEMPERATURE     = 0.0      # no randomness
TOP_P           = 0.0      # nucleus sampling off
TOP_K           = 1        # only highest-probability token
```

```
MAX_TOKENS      = 2048     # response length cap
LOW_CONF_THRESH = 5.0      # threshold for credibility flags
}
```

- **Zero Temperature (TEMPERATURE=0.0)**
  Forces the model to choose the single most likely next token at each step, removing stochastic variation.
- **Nucleus Sampling Off (TOP_P=0.0) and Top-K Restriction (TOP_K=1)**
  Further constrain the sampling distribution so that only the top candidate is ever considered.
- **Fixed Maximum Tokens (MAX_TOKENS)**
  Ensures responses stay within predictable length bounds.
- **Optional Seeding**
  When provided, the seed parameter is passed to the API call, offering an additional layer of reproducibility.

This configuration—coupled with consistent prompt formatting and JSON-mode responses—lays the groundwork for a fully deterministic reasoning pipeline. Let me know when you're ready to dive into the Receptor module and caching strategy.

3.2 Receptor Module: llm_call with LRU Caching

The Receptor module serves as the gateway between raw prompts and structured LLM invocations. Its centerpiece is the llm_call function, which wraps every OpenAI API request with consistent parameters, JSON-mode output, logging, and—crucially—an LRU cache to avoid redundant calls.

```
@lru_cache(maxsize=128)
def llm_call(prompt: str, seed: int = None) -> str:
    """
    Call the OpenAI chat API with JSON-mode, deterministic sampling,
    and optional seed for reproducibility.
    """
    print(f"[LLM CALL] Prompt hash: {hashlib.sha256(prompt.encode()).hexdigest()[:8]}")
    try:
        params = {
            "model": MODEL_NAME,
            "messages": [{"role": "user", "content": prompt}],
            "temperature": TEMPERATURE,
            "top_p": TOP_P,
            "max_tokens": MAX_TOKENS,
            "response_format": {"type": "json_object"}
        }
        if seed is not None:
            params["seed"] = seed
        resp = client.chat.completions.create(**params)
        content = resp.choices[0].message.content.strip()
        print("[LLM CALL]  Response received")
        return content
    except OpenAIError as e:
        print(f"[LLM CALL]  API error: {e}")
        return ""
```

We decorate llm_call with lru_cache(maxsize=128), which means that up to 128 unique (prompt, seed) combinations and their responses are stored in memory—so if the same prompt is used again, the result is retrieved instantly rather than triggering a new API request, both reducing cost and modeling how repeated stimuli shouldn't require re-analysis.

By fixing temperature=0.0, top_p=0.0, and top_k=1, we force the model to choose only the single highest-probability token at each decoding step, thereby guaranteeing that identical prompts always yield identical outputs—a necessity for reproducible academic experiments.

We specify "response_format": "type": "json_object" in every call, compelling the LLM to emit strictly valid JSON; this ensures that downstream code can safely parse claims, reason lists, and score objects without brittle string-processing hacks.

Before each API request, we log a truncated SHA-256 hash of the prompt, giving us a succinct identifier in the logs that makes it easy to trace exactly which input produced which output when debugging or auditing results.

Finally, we wrap the API invocation in a try/except block that catches OpenAIError (and other exceptions), prints a clear error message, and returns an empty string—so that transient network or service hiccups don't crash the entire pipeline, and any failures are recorded for later inspection.

3.3 Unconsciousness Module: Two-Level MFQ Scheduler

To simulate the human tendency to spotlight new information before allowing it to recede into background processing, we implement a minimal two-level feedback-queue scheduler in the MFQScheduler class. In the constructor, two deque objects named high and low are instantiated to represent tasks currently in conscious focus versus those relegated to background "unconscious" processing.

When a new analysis job (such as validating a text chunk) is created, the enqueue(task, high_priority=True) method appends the task to the high queue if it merits immediate attention, or to the low queue otherwise. This explicit prioritization mirrors how our cognitive system flags important stimuli for prompt consideration.

The core of the scheduler, run_all(), repeatedly drains the high queue first:

```
while self.high or self.low:
    task = self.high.popleft() if self.high else self.low.popleft()
    result = task()
    yield result
    self.low.append(task)
```

By always selecting from high when available, we model a "quantum jump" of attention to newly enqueued tasks. Immediately after execution, each task is demoted into the low queue which emulates the fading of conscious focus as tasks transition back into background processing.

Because run_all() yields each task's result as it completes, the scheduler integrates seamlessly with the Consciousness module: each CRIT subroutine invocation becomes a scheduled job that

can be inspected, re-evaluated, or debugged independently. This simple, transparent design captures essential properties of cognitive attention without the complexity of learned policies or heavy infrastructure.

3.4 CRIT Subroutines

The CRIT subroutines implement the core "Consciousness" operations, turning each document chunk into a structured analysis. The following subsections detail how each step is performed in code and essential for transparent, reproducible reasoning.

3.4.1 Claim Extraction. We isolate the primary assertion of each text chunk using the extract_claim(document, seed) function. Internally, this sends a JSON-mode prompt:

```
prompt = (
    "Please respond ONLY with valid JSON: {\"claim\": \"<text>\"}.\n"
    "Identify the single main claim in the document below.\n\n"
    f"{document}\n\nJSON:"
)
```

By incrementing the global LLM_CALL_COUNT and parsing the returned JSON, we reliably retrieve the claim string. This explicit JSON constraint ensures our pipeline never misinterprets free-form text as the claim, while alignment with a fixed seed (if provided) guarantees the same chunk always yields the same extracted claim.

3.4.2 Supporting and Counter-Reason Generation. Next, we generate arguments both for and against the extracted claim via extract_reasons(document, claim, support, seed). Depending on the support boolean, the function crafts a prompt such as:

```
"List the supporting reasons for the claim below as a JSON array of strings.\n"
f"Claim: {claim}\n\n{document}\n\nSupportingReasons:"
```

We again count each LLM invocation and attempt to parse the JSON array. If parsing fails, a line-by-line fallback extracts bulleted text into reason strings. This two-pronged approach captures diverse argument types while maintaining structured output for downstream scoring.

3.4.3 Reason Validation and Credibility Scoring. Each reason is then assessed by validate_reason(reason, claim, seed), which prompts the model to assign both a score (0–10 support strength) and credibility (0–10 trustworthiness) in JSON format:

```
"You are scoring support for the claim on a 0–10 scale (0=no support, 10=strongest support) "
"and credibility on a 0–10 scale (0=not credible, 10=highly credible). "
"Respond ONLY with JSON: {\"score\": <0–10>, \"credibility\": <0–10>}.\n\n"
f"Claim: {claim}\nReason: {reason}"
```

After parsing and clamping these values, we compare credibility against LOW_CONF_THRESH and increment LOW_CONF_FLAG_COUNT for any low-confidence flags. This quantification step converts qualitative arguments into numeric evidence, enabling consistent comparison and aggregation.

3.4.4 Gamma Computation. Finally, the compute_gamma(sup_scores, sup_creds, cnt_scores, cnt_creds) function synthesizes the collected evidence into a single metric gamma:

```
total_sup = sum(s*c for s,c in zip(sup_scores, sup_creds))
total_cnt = sum(s*c for s,c in zip(cnt_scores, cnt_creds))
gamma     = (total_sup - total_cnt) / (sum(sup_creds) + sum(cnt_creds)) if total_cred else 0.0
```

$$\text{total\_sup} = \sum_i s_i\, c_i, \quad \text{total\_cnt} = \sum_j t_j\, d_j$$

$$\gamma = \begin{cases} \dfrac{\sum_i s_i\, c_i \;-\; \sum_j t_j\, d_j}{\sum_i c_i \;+\; \sum_j d_j} & \text{if } \sum_i c_i + \sum_j d_j \neq 0, \\ 0 & \text{otherwise.} \end{cases}$$

$$\gamma = \frac{\sum_i s_i\, c_i \;-\; \sum_j t_j\, d_j}{\sum_i c_i \;+\; \sum_j d_j}.$$

By weighting each support or counter score by its credibility, gamma reflects the net balance of credible evidence. A positive gamma indicates overall support, whereas a negative gamma denotes opposition. This scalar summary affords a concise, interpretable outcome for each chunk, ready for reporting and further analysis.

3.5 Integration: crit_validate End-to-End Workflow

The crit_validate function orchestrates the entire CoCoMo pipeline by first resetting global counters and marking the overall start time to ensure each run begins from a clean state and timing measurements are accurate:

```
global LLM_CALL_COUNT, LOW_CONF_FLAG_COUNT
LLM_CALL_COUNT = 0
LOW_CONF_FLAG_COUNT = 0
overall_start = time.time()
print("=== Starting multi-claim CRIT pipeline ===\n")
```

Next, the function attempts to detect explicitly numbered claim headers in the document; if any are found, it splits the text into labeled chunks accordingly, otherwise it falls back to paragraph-based segmentation—ensuring flexibility in handling both structured and free-form inputs:

```
claim_headers = re.findall(r"\*\*(Claim\s+\d[^\*]+)\*\*", document)
sections = re.split(r"\*\*Claim\s+\d[^\*]+\*\*", document)

if claim_headers:
```

```
    # Build one chunk per numbered claim
    for i, header in enumerate(claim_headers, start=1):
        text = sections[i].strip()
        chunks.append((header, text))
    tail = sections[len(claim_headers)+1].strip() if
    ↪ len(sections) > len(claim_headers)+1 else ""
    chunks.append(("Overall Assertion", tail))
else:
    paras = re.split(r"\n\s*\n", document)
    chunks = [(f"Paragraph #{i+1}", p.strip()) for i, p in
    ↪ enumerate(paras) if p.strip()]
```

With the text segmented into manageable chunks, crit_validate constructs an MFQScheduler and enqueues each chunk's CRIT job at high priority, thereby modeling the initial burst of conscious attention for every new analysis task:

```
scheduler = MFQScheduler()
for label, chunk_text in chunks:
    print(f" Enqueuing CRIT for {label}")
    scheduler.enqueue(lambda txt=chunk_text:
    ↪ _crit_chunk(txt, seed), high_priority=True)
```

The scheduler then executes all queued jobs in order of priority, yielding each chunk's report as it completes; this streaming execution allows intermediate results to be handled incrementally and keeps the pipeline responsive:

```
reports = []
for report in scheduler.run_all():
    reports.append(report)
```

Once all chunks are processed, the function computes aggregate metrics—total elapsed time, counts of support versus opposition, average and mean-absolute gamma scores, as well as logging the number of LLM calls and low-credibility flags—to provide a comprehensive summary of the reasoning session:

```
total_time = time.time() - overall_start
gammas     = [r["gamma_score"] for r in reports]
n_support  = sum(1 for g in gammas if g > 0)
n_opp      = sum(1 for g in gammas if g < 0)
avg_gamma  = sum(gammas)/len(gammas) if gammas else 0
mean_abs   = sum(abs(g) for g in gammas)/len(gammas) if
↪ gammas else 0

print("\n=== CRIT pipeline summary ===")
print(f"• Chunks processed     : {len(gammas)}")
print(f"• Total LLM calls      : {LLM_CALL_COUNT}")
print(f"• Low-cred flags       : {LOW_CONF_FLAG_COUNT}")
print(f"• Total time           : {total_time:.2f}s")
print(f"• -scores (min. . . max)   : {min(gammas):.2f} . . .
↪ {max(gammas):.2f}")
print(f"             mean      : {avg_gamma:.2f}")
print(f"             mean||    : {mean_abs:.2f}")
print(f"• Support vs Opposition: {n_support} vs {n_opp}\n")
```

Finally, function returns the list of per-chunk reports, allowing downstream code or users to inspect detailed claim, reason, and gamma score data for further analysis or visualization.

4 Experimental Evaluation and Analysis

4.1 Case Study (Vertical Farming Document)

As an example, we evaluated CoCoMo on a richly annotated "vertical farming" text that includes explicit Claim 1, Claim 2, and an Overall Assertion, enabling us to test both structured and free-form splitting logic. In code, this document is assigned to the doc variable, and passed verbatim into crit_validate:

```
doc = """
In recent years, the rise of vertical farming has been
↪ heralded...
Claim 1: Vertical farms drastically improve land-use
↪ efficiency.
• Supporting Reason A: By stacking multiple growing
↪ layers...
• Counter-Reason C: The need for artificial climate
↪ control...
Claim 2 (nested): Urban vertical farms contribute
↪ meaningfully...
• Supporting Reason D: A Life-Cycle Assessment (LCA) by
↪ GreenMetrics...
• Counter-Reason E: The same LCA noted that if fossil-fuel
↪ electricity...
Overall Assertion: While vertical farming holds great
↪ promise...
"""
results = crit_validate(doc, seed=42)
```

We deliberately selected the "Claim 1: Quantum-powered irrigation can increase crop yields by "1,000×" example to stress-test CoCoMo's entire workflow:

- **Explicit Claim Headers** The presence of "Claim 1" and "Claim 2" triggers the header-based splitter (1A path in crit_validate), ensuring we exercise our primary chunking logic without falling back to paragraph splits.
- **Outlandish Numbers** Astronomical multipliers ("1,000× yields" and "4 GW from algae") force the LLM to weigh extreme supports against realistic counters, probing its ability to generate and validate both sides.
- **Contradictory Field-Trial Reports** Direct conflicts in the text ("1,000×" vs. "2–3×", "4 GW" vs. "50 kW") stimulate robust counter-reason extraction, verifying that extract_reasons(..., support=False) captures genuine opposition arguments.
- **Technical Jargon** Terms like "quantum entanglement irrigation," "photosynthetic amplification catalysts," and "MFQ-X scheduler" challenge our JSON-mode prompts and parsing routines, confirming the pipeline tolerates domain-specific vocabulary.
- **Mixed Success Narratives** Juxtapositions of "negligible energy cost" with "eutrophication risk," and "sub-ms latency" with "27% starvation," ensure that both supporting and opposing reason sets are non-empty, validating end-to-end coverage of the CRIT subroutines.

4.2 Results Analysis

During the reason-validation step, our logs show exactly where credibility dropped below threshold:

```
   Validating reason snippet: "Field trials in Nevada and
   ↪ the Gobi Desert showed yield multipliers from 50x up
   ↪ to a truly staggering 1,000x compared to standard
   ↪ drip irrigation."
    [DEBUG] current LOW_CONF_FLAG_COUNT = 0
    [DEBUG] LOW_CONF_THRESH         = 7.0
[LLM RAW VALIDATION OUTPUT] '{"score":5,"credibility":4.0}'
   Credibility 4.0 < threshold 7.0, incrementing
   ↪ LOW_CONF_FLAG_COUNT
    [DEBUG] NEW LOW_CONF_FLAG_COUNT = 1
```

After this representative execution, the following report was automatically generated:

Table 1: Processing and Support Metrics

Evaluation Metric	Value
Chunks processed	3
Total LLM calls	9
Low-confidence flags	2
Per-chunk γ-scores	
Claim 1	2.35 (moderate support)
Claim 2	−1.05 (weak opposition)
Overall Assertion	1.47 (moderate support)
Support vs. Opposition	2 vs. 1
Average γ	0.92
Mean $\lvert\gamma\rvert$	1.62

Overall, these results showcase CoCoMo's ability to perform transparent, reproducible critical reasoning. Qualitatively, it efficiently manages API usage, surfaces credibility uncertainties, balances competing arguments quantitatively, and maintains interpretability at each step.

5 Conclusion and Future Work

In this paper, we have presented CoCoMo, a functionalist, reproducible pipeline that simulates core aspects of human "System-2" reasoning using large language models. By decomposing the task into four distinct modules—Receptor, Unconsciousness, Consciousness, and Effector—and enforcing deterministic sampling, JSON-mode output, and prompt caching, CoCoMo delivers transparent, audit-friendly analyses of complex texts. Our novel integration of a two-level feedback-queue scheduler with structured CRIT subroutines (claim extraction, reason generation, validation, and -score computation) enables nuanced balancing of support and counter-evidence.

Through our vertical farming and quantum-irrigation case studies, we demonstrated that CoCoMo:

- Efficiently manages API usage, requiring fewer LLM calls via LRU caching.
- Reliably surfaces low-confidence diagnostics for ambiguous claims.
- Interpretably summarizes net evidence via weighted -scores that align with human intuition.

Despite these strengths, CoCoMo has limitations. It relies on the quality of pre-trained LLMs and may inherit their biases or hallucinations. The rigid JSON-mode prompts can sometimes constrain the richness of generated reasoning, and our two-level scheduler lacks the adaptability of learned attention policies.

Future work will explore several directions:

- Adaptive Scheduling: Integrating learned prioritization such as reinforced learning to replace or augment the MFQ scheduler for more dynamic attention control.
- Expanded Cognitive Functions: Incorporating emotion modeling and exploratory counterfactual modules to deepen empathetic and creative reasoning.
- Multimodal Inputs: Extending CoCoMo to handle images, tables, or numeric datasets alongside text, broadening its applicability.

By combining cognitive theory with deterministic LLM engineering, CoCoMo represents a step toward AI systems that are not only powerful in language understanding but also transparent, verifiable, and aligned with human values. We envision CoCoMo serving as a foundation for future ethical AI frameworks that require rigorous, auditable reasoning pipelines.

Acknowledgments

To Professor Ching-Yung Lin and TAs Zelin Yu and Likhith Ayinala for lecturing and explaining the concepts throughout the course.

References

[1] Shailee Bhatia, Vatsal Mittal, and Saransh Sabharwal. 2024. Building Trust and Transparency in AI: A Review of Explainable AI and its Ethical Implications. In *2024 International Conference on Emerging Technologies and Innovation for Sustainability (EmergIN)*. 650–655. doi:10.1109/EmergIN63207.2024.10961081

[2] Dejan Grubisic, Volker Seeker, Gabriel Synnaeve, Hugh Leather, John Mellor-Crummey, and Chris Cummins. 2024. Priority sampling of large language models for compilers. In *Proceedings of the 4th Workshop on Machine Learning and Systems*. 91–97.

[3] Luz Enith Guerrero, Luis Fernando Castillo, Jeferson Arango-Lopez, and Fernando Moreira. 2025. A systematic review of integrated information theory: a perspective from artificial intelligence and the cognitive sciences. *Neural Computing and Applications* 37, 11 (2025), 7575–7607.

[4] Wenjie Huang, Angelo Cangelosi, and Antonio Chella. 2023. An incremental cognitive architecture of consciousness with global workspace theory. *Available at SSRN 4462597* (2023).

[5] Yashas Samaga, Varun Yerram, Chong You, Srinadh Bhojanapalli, Sanjiv Kumar, Prateek Jain, and Praneeth Netrapalli. 2024. HiRE: High Recall Approximate Top-k Estimation for Efficient LLM Inference. *CoRR* (2024).

[6] Herbert Ullrich, Tomáš Mlynář, and Jan Drchal. 2025. Claim Extraction for Fact-Checking: Data, Models, and Automated Metrics. *arXiv preprint arXiv:2502.04955* (2025).

[7] Zheyuan Wang, Chen Liu, and Matthew Gombolay. 2022. Heterogeneous graph attention networks for scalable multi-robot scheduling with temporospatial constraints. *Autonomous Robots* 46, 1 (2022), 249–268.

[8] Jiali Wu, Hao Dai, Jiashu Wu, Wenming Jin, and Yang Wang. 2024. QMLFQ: A Quantization-based Queue Scheduling Framework for Efficient LLM Serving. In *2024 IEEE Smart World Congress (SWC)*. 1336–1343. doi:10.1109/SWC62898.2024.00208

A Google Colab Code

Below is a part of the core code of CoCoMo in ipynb:

```
# −*− coding: utf−8 −*−

# Step 1: Install & Import Dependencies
print("Step␣1:␣Installing␣dependencies...")

!pip install openai python−dotenv −−quiet

import os, json, hashlib, re, time
from functools import lru_cache
from collections import deque
from openai import OpenAI, OpenAIError
from typing import List, Dict

# Step 2: Initialize OpenAI Client with Deterministic Parameters
print("Step␣2:␣Initializing␣OpenAI␣client")

client = OpenAI(api_key="XXX")

# Global settings
MODEL_NAME = "gpt−4o−mini−2024−07−18"
TEMPERATURE = 0.0
TOP_P = 0.0 # nucleus sampling off for determinism
TOP_K = 1 # restrict to highest−probability token
MAX_TOKENS = 2048
LOW_CONF_THRESH = 5.0

LLM_CALL_COUNT = 0
LOW_CONF_FLAG_COUNT = 0

print(f"OpenAI␣client␣initialized␣with␣model␣{MODEL_NAME}")

# Step 3: Receptor  LLM Call with Caching & JSON Mode
@lru_cache(maxsize=128) # cache up to 128 unique prompts
    :contentReference[oaicite:7]{index=7}
def llm_call(prompt: str, seed: int = None) −> str:
    """
    Call the OpenAI chat API with JSON−mode, deterministic
        sampling,
    and optional seed for reproducibility
        :contentReference[oaicite:8]{index=8}.
    """
    print(f"[LLM␣CALL]␣␣Sending␣prompt␣to␣{MODEL_NAME}␣
        (max␣{MAX_TOKENS}␣tokens,␣
        temp={TEMPERATURE})")
    print(f"[LLM␣CALL]␣Prompt␣hash:␣
        {hashlib.sha256(prompt.encode()).hexdigest()[:8]}")
    try:
        params = {
            "model": MODEL_NAME,
            "messages": [{"role": "user", "content": prompt}],
            "temperature": TEMPERATURE,
            "top_p": TOP_P,
            "max_tokens": MAX_TOKENS,
            "response_format": {"type": "json_object"}
        }
        if seed is not None:
            params["seed"] = seed # best−effort deterministic
                sampling
        resp = client.chat.completions.create(**params)
        content = resp.choices[0].message.content.strip()
        print("[LLM␣CALL]␣␣Response␣received")
        return content
    except OpenAIError as e:
        print(f"[LLM␣CALL]␣␣API␣error:␣{e}")
        return ""
    except Exception as e:
        print(f"[LLM␣CALL]␣␣Unexpected␣error:␣{e}")
        return ""
    return ""

# Step 4: Unconsciousness  Simple Two−Level Feedback Queue
    Scheduler
class MFQScheduler:
    def __init__(self):
        self.high = deque() # high−priority queue
        self.low = deque() # low−priority queue

    def enqueue(self, task, high_priority=False):
        (self.high if high_priority else self.low).append(task)

    def run_all(self):
        # Always drain high queue first (quantum jump)
        while self.high or self.low:
            task = self.high.popleft() if self.high else
                self.low.popleft()
            result = task()
            # After run, demote to low queue for re−evaluation
                next time
            # (simulates fading out of consciousness)
            yield result

# Step 5: CRIT Subroutines (Claim, Reasons, Validation,
    Aggregation)
def extract_claim(document: str, seed: int = None) −> str:
    print("␣Extracting␣claim␣with␣JSON␣mode")
    prompt = (
        "Please␣respond␣ONLY␣with␣valid␣JSON:␣{\"claim\":␣
            \"<text>\"}.\n"
        "Identify␣the␣single␣main␣claim␣in␣the␣document␣
            below.\n\n"
        f"{document}\n\nJSON:"
```

```
    )

    global LLM_CALL_COUNT
    LLM_CALL_COUNT += 1

    out = llm_call(prompt, seed)
    print(f"[LLM RAW CLAIM OUTPUT] {out!r}")
    try:
        claim = json.loads(out)["claim"]
    except:
        claim = out.strip()
    print(f" Claim: {claim}\n")
    return claim

def extract_reasons(document: str, claim: str, support: bool =
     True, seed: int = None) -> list[str]:
    tag = "supporting" if support else "counter"
    print(f" Extracting {tag} reasons for the claim...")
    prompt = (
        f"List the {tag} reasons for the claim below as a JSON
            array of strings.\n"
        f"Claim: {claim}\n\n{document}\n\n"
        f"{tag.capitalize()}Reasons:"
    )

    global LLM_CALL_COUNT
    LLM_CALL_COUNT += 1
    out = llm_call(prompt, seed)
    print(f"[LLM RAW {tag.capitalize()} OUTPUT] {out!r}")
    # Parse JSON array
    try:
        data = json.loads(out)
        key = next((k for k in data if
            k.lower().endswith("reasons")), None)
        reasons = data[key] if key else []
        # reasons = data.get(f"{tag}Reasons") or
            next(iter(data.values()))
    except Exception:
        reasons = [line.strip(" -") for line in out.splitlines() if
            line.strip()]

    print(f" Found {len(reasons)} {tag} reason(s): {reasons}\n")
    return reasons

def validate_reason(reason: str, claim: str, seed: int = None) ->
     tuple[float, float]:
    global LLM_CALL_COUNT, LOW_CONF_FLAG_COUNT
    print(f" Validating reason snippet: {reason[:80]}")
    print(f"    [DEBUG] current LOW_CONF_FLAG_COUNT = "
        {LOW_CONF_FLAG_COUNT}")
    print(f"    [DEBUG] LOW_CONF_THRESH        = "
        {LOW_CONF_THRESH}")
    prompt = (
        "You are scoring support for the claim on a 010 scale "
            (0=no support, 10=strongest support) "
        "and credibility on a 010 scale (0=not credible, "
            10=highly credible). "
        "Respond ONLY with JSON: {\"score\": <010>, "
            \"credibility\": <010>}.\n\n"
        f"Claim: {claim}\nReason: {reason}"
    )

    LLM_CALL_COUNT += 1
    raw = llm_call(prompt, seed)
    print(f"[LLM RAW VALIDATION OUTPUT] {raw!r}")
    try:
        data = json.loads(raw)
        score = max(0.0, min(10.0, float(data.get("score",0))))
        cred = max(0.0, min(10.0,
            float(data.get("credibility",0))))
    except Exception as e:
        print(f" Parse error: {e}")
        score, cred = 0.0, 0.0

    if cred < LOW_CONF_THRESH:
        print(f" Credibility {cred} < threshold
            {LOW_CONF_THRESH}, incrementing
            LOW_CONF_FLAG_COUNT")
        LOW_CONF_FLAG_COUNT += 1
        print(f"    [DEBUG] NEW LOW_CONF_FLAG_COUNT
            = {LOW_CONF_FLAG_COUNT}")

    print(f" Parsed: score={score}, credibility={cred}\n")
    return score, cred

def compute_gamma(sup_scores, sup_creds, cnt_scores,
     cnt_creds) -> float:
    print(" Computing final CRIT score (Gamma)")
    total_sup = sum(s*c for s,c in zip(sup_scores, sup_creds))
    total_cnt = sum(s*c for s,c in zip(cnt_scores, cnt_creds))
    count = len(sup_scores) + len(cnt_scores)

    total_cred = sum(sup_creds) + sum(cnt_creds)
    gamma = (total_sup - total_cnt) / total_cred if total_cred
        else 0.0

    print(f" Gamma raw value: {gamma}\n")
    return gamma

# Step 6: End-to-End CRIT Validation with MFQ Scheduling
def crit_validate(document: str, seed: int = None) -> list[dict]:
    global LLM_CALL_COUNT, LOW_CONF_FLAG_COUNT
    LLM_CALL_COUNT = 0
```

```
LOW_CONF_FLAG_COUNT = 0

print("=== Starting multi-claim CRIT pipeline ===\n")
overall_start = time.time()

# --- BEGIN REPLACEMENT BLOCK ---
# 1A. Try to split on explicit Claim headers
claim_headers = re.findall(r"\*\*(Claim\s+\d[^\*]+)\*\*",
    document)
sections = re.split( r"\*\*Claim\s+\d[^\*]+\*\*", document)

if claim_headers:
    # Build one chunk per numbered claim
    chunks = []
    for i, header in enumerate(claim_headers, start=1):
        text = sections[i].strip()
        chunks.append((header, text))
    # Add the remainder as "Overall Assertion"
    tail = sections[len(claim_headers)+1].strip() \
        if len(sections) > len(claim_headers)+1 else ""
    chunks.append(("Overall Assertion", tail))

else:
    # 1B. FALLBACK: split the doc into paragraphs
    paras = re.split(r"\n\s*\n", document)
    # label each paragraph for debugging
    chunks = [(f"Paragraph #{i+1}", p.strip())
              for i, p in enumerate(paras) if p.strip()]
# --- END REPLACEMENT BLOCK ---

# 2. Schedule each chunks CRIT job
scheduler = MFQScheduler()
for label, chunk_text in chunks:
    print(f" Enqueuing CRIT for {label}")
    scheduler.enqueue(lambda txt=chunk_text:
        _crit_chunk(txt, seed),
                      high_priority=True)

# 3. Run jobs and collect
reports = []
for report in scheduler.run_all():
    reports.append(report)

total_time = time.time() - overall_start
gammas = [r["gamma_score"] for r in reports]
n = len(gammas)
n_support = sum(1 for g in gammas if g > 0)
n_opp = sum(1 for g in gammas if g < 0)
avg_gamma = sum(gammas)/n if n else 0
mean_abs = sum(abs(g) for g in gammas)/n if n else 0

print("\n=== CRIT pipeline summary ===")
print(f" Chunks processed    : {n}")
print(f" Total LLM calls     : {LLM_CALL_COUNT}")
print(f" Low-cred flags      :
    {LOW_CONF_FLAG_COUNT}")
print(f" Total time          : {total_time:.2f}s")
print(f" -scores (minmax)    : {min(gammas):.2f}
    {max(gammas):.2f}")
print(f"          mean       : {avg_gamma:.2f}")
print(f"          mean|.|    : {mean_abs:.2f}")
print(f" Support vs Opposition: {n_support} vs {n_opp}\n")

print("\n=== Multi-claim CRIT pipeline complete ===\n")
return reports

def _crit_chunk(document: str, seed: int) -> dict:
    print("  [Chunk Job] Starting CRIT on document chunk\n")
    start = time.time()

    # 1. Claim
    claim = extract_claim(document, seed)

    # 2. Reasons
    supports = extract_reasons(document, claim, support=True,
        seed=seed)
    counters = extract_reasons(document, claim, support=False,
        seed=seed)
    if not supports: print("  [Chunk Job] No supporting reasons
        extracted!")
    if not counters: print("  [Chunk Job] No counter reasons
        extracted!")

    # 3. Validation
    sup_scores, sup_creds = [], []
    print("  [Chunk Job] Validating supporting reasons")
    for idx, reason in enumerate(supports, 1):
        s, c = validate_reason(reason, claim, seed)
        sup_scores.append(s); sup_creds.append(c)
        if c < LOW_CONF_THRESH:
            print(f" Low credibility ({c}) for supporting reason
                #{idx}")

    cnt_scores, cnt_creds = [], []
    print("  [Chunk Job] Validating counter reasons")
    for idx, reason in enumerate(counters, 1):
        s, c = validate_reason(reason, claim, seed)
        cnt_scores.append(s); cnt_creds.append(c)
        if c < LOW_CONF_THRESH:
            print(f" Low credibility ({c}) for counter reason
                #{idx}")

    # 4. Gamma computation + interpretation
```

```
    gamma = compute_gamma(sup_scores, sup_creds,
        cnt_scores, cnt_creds)

    # 5. Interpretation
    print("␣Gamma␣Interpretation:")
    if gamma>= 8: msg="Very␣strong␣support"
    elif gamma>= 2: msg="Moderate␣support"
    elif gamma>= 0: msg="Weak␣support"
    elif gamma>=−2: msg="Weak␣opposition"
    elif gamma>=−8: msg="Moderate␣opposition"
    else: msg="Very␣strong␣opposition"
    print(f"␣␣␣=␣{gamma:.2f}␣␣{msg}")

    duration = time.time() − start
    print(f"␣Chunk␣completed␣in␣{duration:.2f}s\n")

    # 6. Report
    report = {
        "claim": claim,
        "supporting_reasons": supports,
        "support_scores": sup_scores,
        "support_credibility": sup_creds,
        "counter_reasons": counters,
        "counter_scores": cnt_scores,
        "counter_credibility": cnt_creds,
        "gamma_score": gamma
    }
    print("\n␣Detailed␣Report:")
    print(json.dumps(report, indent=2))
    return report

# Step 7: Smoke Test
print("␣Running␣smoke␣test␣with␣sample␣document...\n")

doc = """
**Claim 1: Quantum−powered irrigation can increase crop yields
    by 1,000**
In 2025, the startup AgriFlux patented a quantum−entanglement
    irrigation system they claim drips water directly into plant
    cell vacuoles. According to their white paper, Field trials in
    Nevada and the Gobi Desert showed yield multipliers from 50
    up to a truly staggering 1,000 compared to standard drip
    irrigation. They allege energy costs were negligible because
    the device harvests ambient cosmic microwave background
    radiation.

Despite these assertions, independent testers observed only a 23
    boost under laboratory conditions, and many trials failed
    outright when humidity was above 20%  a curious side−effect
    the company glosses over.

−−−

**Claim 2: Urban algae farms will power entire city grids by 2030**
A consortium of five technology firms published in *Futuristic
    Energy Monthly* that rhodophyte bioreactors installed on
    every skyscraper faade in Tokyo will generate 4 GW of peak
    power, enough to offset 30% of the citys nighttime grid load.
    They cite proprietary photosynthetic amplification catalysts
    that allegedly boost algal photon capture by 700%.

However, panelists at the 2024 GreenTech Summit noted that
    scale−up tests only ever produced ~50 kW in total from a 100
    m installation  orders of magnitude below projections. Critics
    also question the environmental impact of continuous
    red−light exposure and potential eutrophication from algae
    runoff.

−−−

**Claim 3: AI−driven MFQ schedulers achieve human−level
    multitasking**
Drawing on the CoCoMo framework, ZephyrAI claims their
    MFQ−X scheduler can juggle 1,024 simultaneous background
    tasks with per−task latency under 1 ms, matching the
    attentional quantum jump capabilities of an average human
    brain. Benchmarks released on GitHub show sub−millisecond
    context−switching across 16 priority levels.

Yet, on PulsarComputes test cluster, real−world workloads (video
    decoding + real−time strategy game AI) caused MFQ−X to
    starve critical threads 27% of the time, leading to frame drops
    and missed network deadlines. The vendors own issue tracker
    logs over 300 priority inversion bug reports in the last quarter.

−−−

**Overall Assertion**
In each of these cutting−edge technologies, the promotional hype
    vastly outpaces reproducible results. While the theoretical
    models (quantum irrigation, urban algae farms, MFQ−X
    scheduling) point to revolutionary gains, every published
    field trial or benchmark shows far more modest
    improvements  if they succeed at all.

"""

results = crit_validate(doc, seed=42)
for idx, rpt in enumerate(results, 1):
    print(f"\n===␣Report␣for␣chunk␣#{idx}␣===")
    print(json.dumps(rpt, indent=2))

import pprint; pprint.pprint(result)
```