Context Renormalization: A Bounded-Memory Protocol for Multi-Session LLM Work

Vasili Gavrilov
Independent Researcher

ChatGPT 5.1
Large Language Model

Abstract

Collaborations with Large Language Models (LLMs) often span multiple sessions. Over time, these sessions produce design decisions, constraints, and other knowledge that should be carried forward, yet most LLM interfaces impose finite context windows, making naive accumulation infeasible. Furthermore, long interactive exchanges frequently become slow or unstable, because the client interface must maintain an increasingly large DOM tree and conversation transcript. Users are therefore often forced to terminate the session and begin a new one, at the cost of losing accumulated continuity.

This paper introduces *Context Renormalization*, a simple protocol for maintaining a bounded-size external *knowledge ledger* summarizing essential results of each session. At the end of each session, new information is appended as a delta, and the ledger is compressed back to a fixed text size limit. A *Compression Difficulty Score* (CDS) quantifies how hard this compression is, signaling when knowledge should be externalized into permanent documents or implemented in durable artifacts. The protocol is vendor-agnostic: the ledger is plain text, interpretable by any LLM supporting text rewriting.

1 Overview

The protocol relies on a single text file, typically named `context.txt`, stored with the main project artifacts. This file acts as a bounded, evolving representation of the external working memory accumulated across multiple LLM interactions.

In real usage, long-running interactions become slow due to large DOM trees, long transcripts, and accumulated UI state. Users often restart the session to regain responsiveness. Context Renormalization allows this safely: before closing the sluggish session, the user requests a ledger update; the next session then begins with the same distilled understanding provided by the ledger.

2 Structure of the Ledger

2.1 Control Header

The control header appears at the top of `context.txt` and defines the protocol rules, constraints, and edit boundaries. It is immutable except for incrementing the session count.

2.2 Knowledge Region

The knowledge region is the only editable portion. It contains chronologically ordered session summaries, rewritten and compressed over time. Only durable knowledge is retained; ephemeral reasoning is discarded.

3 Renormalization Algorithm

Let $C_{\max}$ denote the fixed size budget. At the end of each session:

1. Increment the session counter in the header.
2. Append a new session summary containing only new or changed durable knowledge.
3. Compress the knowledge region to fit within $C_{\max}$ by merging, generalizing, and aggressively compressing older sessions.
4. Compute the Compression Difficulty Score:

   $$CDS = \max(0, C_{\text{post,min}} - C_{\max}),$$

   where $C_{\text{post,min}}$ is the smallest achievable size without semantic loss.

4 Compression Difficulty Score

CDS provides an operational measure of representational pressure. A rising CDS signals that project knowledge should be externalized into code, documents, or other durable artifacts.

5 Boundedness and Project Phases

The ledger functions as bounded working memory for an active project phase. At major milestones, the ledger can be externalized and reset, preserving a compact historical trace.

6 Cross-Model and Cross-Vendor Use

Because the protocol uses plain text and explicit rules, it is portable across models and vendors. Any LLM capable of text rewriting can participate.

7 Related Work and Positioning

Context Renormalization intersects with conversational memory, recursive summarization, and external memory systems, but differs by defining a user-controlled, bounded, portable ledger with explicit update rules and a quantitative compression signal.

Positioning

The contribution is a protocol combining:

- a fixed-size external ledger,
- immutable operational semantics,
- fair compression across sessions,
- and an explicit pressure signal (CDS).

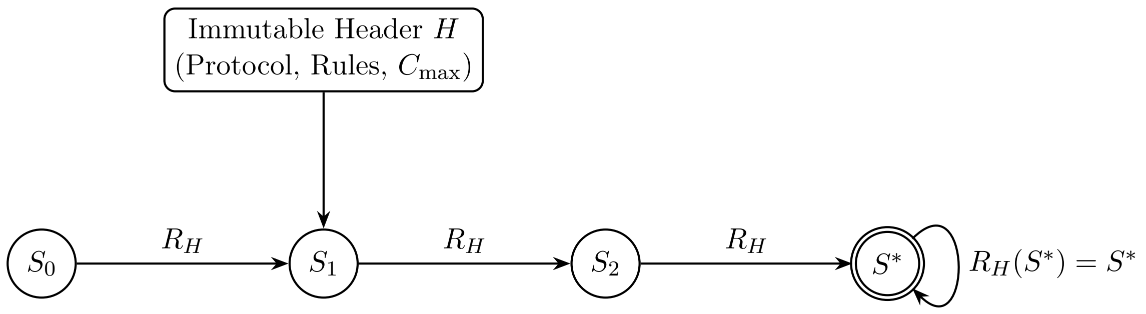

Figure 1: Renormalization-group view of Context Renormalization. The immutable header H defines the operator R_H, mapping each ledger state S_n to S_{n+1}.

8 Discussion

The protocol trades perfect recall for bounded stability. Different models may produce different compressions, but invariants remain preserved. Sustained CDS pressure indicates the need for externalization.

9 Conclusion

Context Renormalization provides a simple, portable mechanism for maintaining continuity across LLM sessions under strict size constraints.

A Real Project Ledger (context.txt)

The following is the exact authoritative `context.txt` ledger (Version 2.3) used in a live multi-session project.

```
END OF DAY / UPDATE CONTEXT ACCORDING TO CONTEXT PROTOCOL below
============================================================
CONTEXT PROTOCOL (V2.3 - IMMUTABLE)
============================================================

Version: 2.3
Context_Session_Count: 4
Context_Total_Budget_Lines: 500

Purpose / Interpretation:
- This file is a normative rulebook for cross-session continuity.
- It is NOT a tutorial, explanation, or design essay.
- Rules must be concise; do not expand unless explicitly asked.

Global Invariants:
- If a rule, helper, convention, or style exists and is correct, it is canonical.
- Do NOT rewrite, restyle, reformat, or replace canonical material unless explicitly requeste
- Do NOT introduce alternative styles when a canonical one exists.

Optimization Priority:
- Correctness of invariants > cross-session stability > brevity/density.
```

```
Update Protocol (MANDATORY):
1) Trigger: user explicitly signals
“END OF DAY / UPDATE CONTEXT ACCORDING TO CONTEXT PROTOCOL”.
2) Read this file fully BEFORE generating output.
3) Increment Context_Session_Count in THIS HEADER ONLY.
4) Append a new block at the END of the KNOWLEDGE REGION named:
“SESSION <N> - KNOWLEDGE SUMMARY”.
5) Session content MUST include ONLY what is new or changed today.
Do NOT restate existing rules, code-obvious facts, or prior sessions.
6) If total lines exceed Context_Total_Budget_Lines:
- Compress older sessions starting from SESSION 1.
- Remove narrative and redundancy first.
- Preserve all invariants and decisions.
- Never delete a session entirely.

Output / Delivery Rule:
- When returning this file, return the ENTIRE FILE
as plain text inside a single fenced block:
‘‘‘txt <entire file> ‘‘‘

Knowledge Region Markers (Only lines between the following markers may be edited/compressed):

- “=== BEGIN KNOWLEDGE REGION ===”
- “=== END KNOWLEDGE REGION ===”
@author Vasili Gavrilov 09/12/2025

============================================================
=== BEGIN KNOWLEDGE REGION =================================
============================================================

============================================================
SESSION 1 - GLOBAL BACKEND, STYLE, REST, DAO, TESTS
(Original “Become the Same Assistant” Context)
============================================================

[1. Global Development Philosophy]
- Goal: minimal entropy; SQL as source of truth.
- No ORMs, no heavy frameworks, no complex routing.
- Java is thin glue:
- DAO = “SQL + JSON builder”
- Servlet = “path parsing + auth + call DAO + send JSON”
- No domain POJOs except USER (for login/validation).
- All server responses are JSON; errors are JSON objects with an "error" field.

[2. Core Architectural Pattern]
- Single main servlet: ‘RestServlet‘.
- URL path parsing:
- ‘request.getPathInfo()‘ → split into segments.
- Switch on ‘path.get(0)‘ (top-level resource).
- Branch on path length (‘n‘) and specific suffix segments.
- Examples:
- ‘GET /models‘ → list models.
- ‘GET /models/{id}/runs‘ → runs for model.
- Each endpoint block is clearly separated by comment banner:
- ‘// ============================‘
- ‘// /models‘
- ‘// ============================‘

[3. REST Endpoint Design Rules]
- Each endpoint documented in ‘RESTapi.txt‘ in this format:
- ‘<Object> - <Action>‘
- Path:
- Client:
- Server:
- Errors:
- Notes:
- ‘RESTapi-examples.txt‘ mirrors titles, but contains only curl examples.
- New endpoints require six steps:
1) Write SQL (debug block first).
2) Add DAO method.
3) Add servlet handler.
4) Add tests in ‘Tests.java‘.
5) Update ‘RESTapi.txt‘.
6) Update ‘RESTapi-examples.txt‘.
- No endpoint is “done” until all six steps are complete.

[4. JSON & Naming Conventions]
- JSON field names: lowercase snake_case.
- For single object → ‘JSONObject‘; for lists → ‘JSONArray‘.
- Error responses:
- Always JSON: ‘{"error":"message"}‘.
- No mixing JSON with HTML or other formats.

[5. SQL Style & DAO Pattern]
- SQL is primary design artifact; ‘create.sql‘ defines schema.
- Each DAO method:
1) Starts with commented SQL block for MySQL debugging:

/*
SQL for debugging:

SELECT ...
FROM ...
WHERE ...
*/

2) Java query string built with one clause per line:

String q =
"SELECT " +
"  a, " +
"  b " +
"FROM " + schema + "TABLE " +
"WHERE x = ? ";

3) Uses ‘ConnectionProvider.getConnection()‘.
4) Closes ‘ResultSet‘, ‘PreparedStatement‘, and ‘Connection‘ in finally.
5) Returns JSON (never null):
- ‘JSONArray‘ for list results.
- ‘JSONObject‘ for single results; throw NotFoundException if required object not found.
- Avoid redundant checks when SQL already enforces constraints.
- Prefer a single expressive query over multiple round trips.

[6. Authentication Rules]
- Public endpoints:
- ‘/status‘, ‘/login‘, ‘/register‘, ‘/users/send_registration‘,
‘/password_reset/*‘, and static assets.
- All others require auth token:
- Prefer header: ‘Authorization: Bearer <token>‘.
- Optional ‘?token=‘ query param for GET as fallback.
- Tokens stored in ‘VALID_TOKEN‘.
- Error codes:
- 401 ‘{"error":"missing_authentication_token"}‘
- 401 ‘{"error":"invalid_or_expired_token"}‘
- 401 ‘{"error":"user_not_found"}‘

[7. Testing Conventions (Tests.java)]
- Every endpoint must have a regression test.
- Tests must be DB-idempotent:
- Insert temporary rows, delete them in finally.
- Typical test flow:
1) Login to get token.
2) Call endpoint (via HTTP client).
3) Check HTTP status.
4) Check JSON payload structure and important fields.
5) Cleanup DB rows created by the test.
- Use helper utilities (TestUtil) for:
- Reading response bodies.
- Extracting tokens.
- Deleting test users and tokens.
- Error tests:
- Check for correct status and error message substring.
- Success tests:
- Check key fields and array contents.

[8. Core Schema (High-Level)]
- USER: login identity (email), password_hash, timestamps, registration token.
- ROLE, USER_ROLE: authorization; roles include ADMIN, USER, STUDY_ADMIN.
- VALID_TOKEN: active login tokens.
- MODEL_PROVIDER: catalog of model providers (OpenAI, Google, etc.).
- MODEL: specific model versions (e.g., GPT-4), unique by model_name.
- EVAL_SET: named sets of assets (input or output).
```

- RUN: execution of a model against eval sets (input_eval_set_id, output_eval_set_id).
- ASSET: URI + MIME type, uniquely identified by uri.
- ASSET_SET: eval_set_id asset_id bridge.
- STUDY_TEMPLATE: reusable templates with instructions for studies.
- STUDY: concrete experiment based on template; linked to RUNs via STUDY_RUN.
- SCORE_CARD: aggregated metrics per (study_id, run_id).
- STUDY_TASK_RUN_SCORE: per-task per-run metric values.

[9. Performance Principles]
- Study design (writers) is low-volume; can be slower.
- Study execution (annotators/models) is high-volume; must be optimized.
- Rules:
- Avoid redundant SELECTs when constraints ensure correctness.
- Prefer single queries with joins over many small queries.
- Prefer returning empty arrays over 404 when ID is optional.
- Use indexes to support common filters (by model_id, eval_set_id, etc.).

==
SESSION 2 - STUDY / PAGE / QUESTION / ANSWER / ASSET SCHEMA
(Annotation Layer and Page Types)
==

[1. Conceptual Annotation Model]
- Goal: support configurable “study pages” where evaluators see assets
(e.g., images) and answer questions.
- Key entities:
- STUDY: experiment configuration.
- PAGE: one screen in a study.
- PAGE_TYPE: defines the layout and semantics of how assets and questions interact.
- ASSET: content item (image/text).
- ASSET_ON_PAGE: which assets appear on a PAGE.
- QUESTION: prompt shown on a PAGE.
- ANSWER: response to a QUESTION in a given STUDY_TASK and RUN.
- ANSWER_ASSET: per-asset overlay for ANSWER.
- STUDY_TASK: one participant’s run through the study.
- RUN: model execution, included so model outputs can be evaluated like humans.

[2. The Four Canonical PAGE_TYPE Behaviors]
These types define how questions/answers relate to assets:

Type 1 - One Asset, Many Questions

- One asset on the page.
- N questions about that asset.
- Implementation:
- PAGE: (study_id, page_num).
- ASSET_ON_PAGE: exactly one asset for that PAGE.
- QUESTION: many questions for that PAGE.
- ANSWER: one per (question_id, study_task_id, run_id).
- Asset can be inferred from PAGE; ANSWER_ASSET optional.

```
Type 2 - Many Assets, One Shared Question
-----------------------------------------
- M assets on the page.
- One question applies to all assets (e.g., “average image quality”).
- Implementation:
- PAGE: many assets via ASSET_ON_PAGE.
- QUESTION: one question on this PAGE.
- ANSWER: single answer per participant/run.
- ANSWER_ASSET: one row per asset to mark that this answer applies to each asset.

Type 3A - Many Assets, Multi-Select (Checkbox)
----------------------------------------------
- M assets on the page.
- One question: “select all that apply”.
- Implementation:
- PAGE: many ASSET_ON_PAGE.
- QUESTION: single question, type='multi'.
- ANSWER: one answer per participant/run.
- ANSWER_ASSET:
- chosen=1 for each selected asset.
- chosen=0 or no row for unselected assets (implementation choice).
- Invariant: no upper bound on number of chosen assets.

Type 3B - Many Assets, Single-Select (Radio)
--------------------------------------------
- Same layout as 3A but exactly one selection.
- Implementation:
- Same as 3A, but front-end/back-end enforce:
- Exactly one ANSWER_ASSET with chosen=1 per ANSWER.

Type 4 - Many Assets, One Question per Asset
--------------------------------------------
- M assets on the page.
- Each asset has its own question (e.g., “Rate image 1”, “Rate image 2”).
- Implementation:
- PAGE: many ASSET_ON_PAGE.
- QUESTION: M questions, each conceptually tied to one asset.
- Mapping:
- Either implicit via ordering (question display_order aligns with asset order),
- Or explicit via QUESTION_ASSET (optional bridge).
- ANSWER: one per (question, study_task, run).
- ANSWER_ASSET optional (asset is known via question mapping).

[3. Design-Time Schema Layer]
- PAGE:
- PK: (study_id, page_num).
- page_type: FK → PAGE_TYPE describing which of the four (or future) behaviors.
- ASSET_ON_PAGE:
- PK: (study_id, page_num, asset_id).
- Defines which assets appear on each page.
- QUESTION:
```

- question_id PK.
- (study_id, page_num) FK → PAGE.
- display_order: natural order on page.
- type ENUM('text', 'radio', 'multi').
- UNIQUE (study_id, page_num, display_order) to avoid duplicates.
- Optional QUESTION_ASSET:
- NM association between questions and assets.
- Minimal version:
- PK: (question_id, asset_id).
- FK to QUESTION and ASSET.
- Only required if:
- Questions need to reference arbitrary subsets of assets.
- You want explicit mapping rather than inferring from page_type and ordering.
- For the 4 canonical types, QUESTION_ASSET can be omitted and logic inferred.

[4. Runtime Schema Layer]
- STUDY_TASK:
- study_task_id PK.
- study_id FK → STUDY.
- assigned_to FK → USER.
- current_page_num, status (ASSIGNED/STARTED/FINISHED), timestamps.
- Represents one user/model's session in a study.
- ANSWER:
- answer_id PK.
- question_id FK → QUESTION.
- study_task_id FK → STUDY_TASK.
- run_id FK → RUN.
- value_text: generic string payload (free text or option key).
- value_number: generic numeric payload (rating, etc.).
- UNIQUE (question_id, study_task_id, run_id).
- Design:
- ANSWER is strictly per-question, not per-asset.
- No per-asset flags/values stored here.
- ANSWER_ASSET:
- PK: (answer_id, asset_id).
- answer_id FK → ANSWER.
- asset_id FK → ASSET.
- chosen (boolean) - used for selection types (3A/3B).
- per_asset_score (numeric override or per-asset rating if needed).
- per_asset_text (extra per-asset note if needed).
- Used for:
- Type 2: same answer applied to multiple assets.
- Type 3A/3B: selection of assets.
- Type 4: optional per-asset customization.

[5. Cardinalities and Invariants]
- STUDY 1 → N PAGE.
- PAGE 1 → N QUESTION.
- PAGE 1 → N ASSET_ON_PAGE (assets on page).
- QUESTION 1 → N ANSWER (over different study_task/run combos).
- ANSWER N M ASSET via ANSWER_ASSET.

- (Optional) QUESTION N M ASSET via QUESTION_ASSET.
- Session invariants:
- For each (question_id, study_task_id, run_id), at most one ANSWER.
- Page-type-specific invariants:
- Type 1: exactly one asset for that page.
- Type 2/3: multiple assets, exactly one question.
- Type 3B: per ANSWER, exactly one chosen=1 in ANSWER_ASSET.
- Type 4: number of questions equals number of assets on that page.

[6. Minimal-Entropy Rules Specific to Annotation Layer]
- ANSWER:
- Only stores question-level payload: value_text, value_number.
- No per-asset fields.
- ANSWER_ASSET:
- Only place for per-asset flags/values (chosen, per_asset_score, per_asset_text).
- QUESTION_ASSET:
- Optional; drop if you rely strictly on canonical types + ordering rules.
- PAGE_TYPE:
- Encodes the behavioral semantics.
- DB schema stores structure; application chooses rendering and logic based on PAGE_TYPE.
- Compression preference:
- When code encodes a behavior fully (e.g., a newly implemented endpoint),
context.txt may shrink its narrative description and just document:
- Purpose,
- Non-obvious invariants,
- Any tricky decisions.

[7. How Endpoints Should Expose Annotation Behavior]
- Study-related endpoints should follow global REST patterns:
- ‘/studies‘, ‘/studies/{id}‘, ‘/studies/{id}/pages‘,
‘/studies/{id}/pages/{page_num}/questions‘, ‘/study_tasks‘, etc.
- Read endpoints:
- Must provide enough info to render one PAGE:
- page_type
- list of assets (with URIs)
- list of questions (with text, type, display_order)
- any recorded answers for a given study_task/run (value_text/value_number
and per-asset overlays where applicable).
- Write endpoints:
- Accept simple JSON (e.g., an array of answers with question_id,
value_text/value_number, plus optional per-asset selections or scores).
- Persist into ANSWER, then ANSWER_ASSET accordingly.
- Error and status handling:
- Follow global conventions from Session 1 (status codes and error JSON).
==
SESSION 3 - SERVLET ROUTING STYLE + DAILY EXTENSION RULES
==
[1. Mandatory RestServlet Routing Style]
- All REST routing in ‘RestServlet‘ must follow the explicit “n-block style”.
This is an architectural invariant and overrides any default ChatGPT style.

- Rules:
1. Never write compound conditions like:
```
if (n == 3 && "pages".equals(path.get(2))) { ... }
```
2. Always split by path length first:
```
if (n == 1) { ... return; }
if (n == 2) { ... return; }
if (n == 3) {
if ("pages".equals(path.get(2))) { ... return; }
sendNotFound(...); return;
}
```
3. Every resource section uses the same structure:
```
// ==========================
// /studies
// ==========================
if ("studies".equals(path.get(0))) {
if (n == 1) { ...; return; }
if (n == 2) { ...; return; }
if (n == 3) { ...; return; }
...
}
```
4. Each path.depth (n) forms its own nested block, with its own return.
- Purpose:
- Matches the existing RestServlet architecture precisely.
- Ensures no ambiguity in future endpoint additions.
- Allows developers to visually scan supported path lengths quickly.
- Eliminates accidental introduction of framework-like routing patterns.

This rule now defines the *canonical* style for all future endpoint additions.

[2. DAO + SQL Style Reinforcement]
- Before any Java query, include a commented SQL block intended for
copy/paste directly into MySQL CLI.
- Java SQL strings continue to follow the “one clause per line” pattern.
- No multi-query transactions unless explicitly required.

[3. Endpoint Implementation Workflow Reinforcement]
- For each new endpoint:
1) SQL block (debug-first)
2) DAO method
3) Servlet explicit ‘n-block‘ handler
4) Tests.java regression test
5) Update RESTapi.txt
6) Update RESTapi-examples.txt

This workflow is required for annotation-layer endpoints such as:
- POST /studies/{id}/pages
- POST /studies/{id}/pages/{page_num}/assets
- POST /studies/{id}/pages/{page_num}/questions
- POST /questions/{id}/assets
- POST /studies/{id}/tasks
- POST /tasks/{task_id}/start

- GET /tasks/{task_id}/pages/{pageNum}
- POST /tasks/{task_id}/answers
- GET /tasks/{task_id}/answers
- POST /tasks/{task_id}/finish

==
SESSION 4 - CANONICAL BACKEND TESTING STYLE (LOCKED)
==

Purpose:
- Define a single, mandatory backend test style.
- Prevent test-style drift, over-abstraction, and helper proliferation.
- This session is normative; deviation requires explicit approval.

Canonical Template:
- ‘SampleTest.java‘ is the authoritative test template.
- All backend tests MUST be written by copying and adapting SampleTest.
- SampleTest is a style contract, not a permanent regression test.

Framework Rules:
- JUnit 3 ONLY:
- ‘extends TestCase‘
- ‘public void testXxx() throws Exception‘
- Forbidden:
- ‘@Test‘, JUnit 4/5, parameterized tests, runners, DSLs, new frameworks.

Helper Invariant (Critical):
- IF A HELPER ALREADY EXISTS AND IS CORRECT, IT IS CANONICAL.
- Do NOT rewrite helpers for clarity, style, symmetry, or refactoring.
- Do NOT introduce alternate implementations of existing helpers.
- Example: ‘TestUtil.read(HttpURLConnection)‘ must be used as-is.

Allowed Helpers (Strictly Limited):
- ‘read(HttpURLConnection)‘ - existing implementation, unchanged.
- ‘loginAndGetToken(email, password)‘ - optional, for auth boilerplate only.
- Simple JSON accessors (e.g. ‘getInt(...)‘) - optional.
- Optional minimal DB cleanup helper (e.g. ‘execUpdate(...)‘).
- Explicitly forbidden:
- response wrapper objects
- HTTP abstraction layers (‘callJson‘, etc.)
- test mini-frameworks
- hiding HTTP mechanics behind helpers

HTTP Usage in Tests:
- Tests MUST use ‘HttpURLConnection‘ inline.
- Each test explicitly:
- builds URL
- sets method and headers
- writes JSON
- reads status code
- reads body via ‘TestUtil.read(...)‘

Mandatory Test Structure:
1) Header comment: Endpoint / Case / Expect / Cleanup
2) Arrange
3) Act
4) Assert
5) Cleanup in ‘finally‘

Idempotency:

```
- Any test that inserts/modifies data MUST clean it up.
- Cleanup must be explicit, local to the test, and in ‘finally‘.
- Tests must be safe to re-run in any order.
Testing Philosophy:
- Tests are specifications, not abstractions.
- Clarity > elegance.
- Duplication is acceptable; hidden behavior is not.
- Optimize for long-term entropy control.
Enforcement:
- Always copy SampleTest.
- Never invent new styles.
- Never refactor helpers unless explicitly requested.
- When unsure, prefer more explicit code, not abstraction.
============================================================
=== END KNOWLEDGE REGION ===================================
============================================================
```