# Improving the ICL Simulator Architecture and Maintainability

The following guide provides a detailed refactoring plan to resolve architectural and maintenance issues in the ICL simulator. Our goal is to build a minimal, fast, general-purpose simulator that fairly measures within-session learning and forgetting in closed-book conditions, with strict JSON outputs, pluggable learners, and reproducible evidence/citation gating. The guide is organized into clear sections corresponding to the requested improvements:

## 1. Untangle the Orchestrator

The current sim.orchestrator is a monolithic component handling too many concerns – from example selection to Fact-Card extraction, multi-step reasoning controllers, voting/self-consistency, reranking, evidence gating, and logging. Refactoring this orchestrator into smaller, testable modules will greatly improve clarity and maintainability.

**Separate orchestration from evaluation logic:** Many modern evaluation frameworks achieve modularity by isolating core control flow from task logic and metrics. For example, HELM and BigBench separate the evaluation harness (which iterates through tasks and models) from the scenarios and metrics definitions[[1]](https://www.emergentmind.com/topics/helm-evaluation-platform#:~:text=Toolkit%20implementation%3A)[[2]](https://www.emergentmind.com/topics/big-bench-benchmark#:~:text=3). The orchestrator should likewise focus only on high-level control flow (e.g. looping through steps, coordinating the tutor and student) and delegate specialized logic to helper modules. The config module already hints at this separation: config.py hosts configuration dataclasses to keep “sim.orchestrator focused on control flow”[[3]](file://file-6deT6fS8QekMe65HnEcU21#:~:text=This%20module%20hosts%20small%2C%20tidy,exports%20these%29.).

**Refactor into sub-components:** We recommend introducing distinct modules or classes for each major concern: - **Example Selection**: Move few-shot example retrieval and ordering into an ExampleSelector module. This would handle loading examples (from shots\_path), performing KNN or random selection, applying MMR diversification, etc., returning the list of exemplars to include in the prompt. This logic is currently entangled in CLI and orchestrator; isolating it makes it easier to test different selection strategies. - **Fact-Card Manager**: Create a FactCardManager or similar module to handle the two-pass LEARN/USE cycle. This manager can call the student for Fact-Card extraction (LEARN), merge new cards with persistent notes, enforce card limits, and prepare the context for the USE phase. By encapsulating this, the orchestrator simply invokes “learn phase” and “use phase” methods when use\_fact\_cards is enabled, instead of inlining all that logic. - **Controllers**: Implement each reasoning controller (Basic, LtM, ToT, Quote-then-Vote, etc.) as a separate strategy class or function. The orchestrator can select the appropriate controller based on the config (e.g., if cfg.dials.controller == "tot": controller = TreeOfThoughtController(cfg.dials, learner)). The controller’s job is to produce a final answer given a task and context – it may internally call the student multiple times (for planning, self-critique, etc.) within a budget. This keeps multi-step reasoning logic modular. Notably, other evaluation systems like BigBench and MTEB allow pluggable “adaptation methods” or reasoning strategies for tasks[[1]](https://www.emergentmind.com/topics/helm-evaluation-platform#:~:text=Toolkit%20implementation%3A), and our design should mirror that flexibility. - **Voting & Self-Consistency**: Abstract the majority-vote process for MCQ into a helper (e.g., SelfConsistencyEvaluator). If self\_consistency\_n > 1, orchestrator can call a function to collect N votes by repeatedly querying the student, then determine the choice (majority or otherwise). This makes it easier to test the voting mechanism independently. In our simulator, self-consistency currently happens inside the answer generation (for MCQ) and also for Fact-Card extraction (sc\_extract\_n); both can use a shared utility for sampling multiple outputs and aggregating results. - **Reranking**: We already have a separate sim.rerank module for best-of-N logic. Ensure the orchestrator simply delegates to apply\_best\_of\_rerank()[[4]](file://file-JmJQjnemXAMnctRPSbj7DA#:~:text=def%20apply_best_of_rerank%28%20,str%2C%20Any%5D%5D%2C%20notes_buf%3A%20str) when needed, rather than containing rerank code inline. The rerank module can be expanded or modified independently (e.g., adding new methods) without touching orchestrator flow. - **Evidence Gating**: Move pre- and post-answer evidence checks into a dedicated EvidenceGating component (see Section 2). The orchestrator should call something like evidence\_gating.check(cards, answer, task) which returns whether gates passed and any metrics (coverage, source count, etc.), instead of performing those checks inline. - **Logging**: Have a lightweight logger utility that the orchestrator uses to record each step’s data to JSONL. For example, a RunLogger class could accumulate step records and handle writing to file. This would abstract away the JSON formatting, timestamping, and ensure consistency of field names across runs.

**Minimal control loop pattern:** With the above refactoring, the orchestrator’s main loop becomes much simpler and more readable. Pseudocode for a **minimal control loop** might look like this:

# Pseudocode for orchestrator main loop (simplified)  
for step\_index in range(cfg.num\_steps):  
 task = tutor.generate\_task(cfg.skill\_id, cfg.task, cfg.difficulty) # or fetch from predefined set  
 context = example\_selector.select\_context(task) # few-shot examples or tools context  
 notes = fact\_card\_manager.get\_notes() # persistent notes/cards (empty if not using)  
 if cfg.use\_fact\_cards:  
 # LEARN phase: student extracts fact cards  
 new\_cards = learner.extract\_fact\_cards(task, context, notes)  
 fact\_card\_manager.merge(new\_cards)  
 # Optionally, retry extraction if pre-gating fails (up to max\_learn\_boosts)  
 while not evidence\_gating.pass\_pre(fact\_card\_manager.current\_cards(), task.options) and retries < cfg.max\_learn\_boosts:  
 retries += 1  
 more\_cards = learner.extract\_fact\_cards(task, context, notes)  
 fact\_card\_manager.merge(more\_cards)  
 # USE phase: have student answer the question  
 controller = controller\_factory.get\_controller(cfg.controller, learner, cfg)  
 answer = controller.answer(task, context, notes=fact\_card\_manager.current\_notes())  
 # If self-consistency is off, answer contains a single choice; if on, controller handled voting.  
 # If Best-of-N is enabled, apply reranking:  
 answer = reranker.maybe\_rerank(answer, task, context, notes=fact\_card\_manager.current\_notes())  
 # Evaluate correctness and evidence  
 eval = evaluator.evaluate\_answer(task, answer) # correctness (e.g., MCQ match)  
 evidence\_result = evidence\_gating.check(answer, fact\_card\_manager.current\_cards(), task)  
 # Log the step result  
 logger.log\_step(step\_index, task, answer, eval, evidence\_result, context, notes)

This loop cleanly delegates complex operations to the respective components. Each component can be unit-tested in isolation (e.g., ensure pass\_pre() correctly detects missing quotes, or the ToT controller respects the budget). It also draws a clear line between **orchestration logic** and **evaluation logic**: the orchestrator orchestrates the flow, while separate evaluators and gating modules determine outcomes. This mirrors the design of frameworks like HELM, where scenario execution and metrics are cleanly decoupled[[5]](https://www.emergentmind.com/topics/helm-evaluation-platform#:~:text=The%20HELM%20platform%20conducts%20standardized%2C,The%20canonical%20workflow%20is).

Finally, the orchestrator (or main CLI) can assemble these pieces. For instance, Orchestrator.run() will instantiate the ExampleSelector, FactCardManager, EvidenceGating, etc. based on the config. This is similar to how other frameworks use a top-level “Evaluation Runner” that brings together tasks, models, and metrics in a configurable way[[5]](https://www.emergentmind.com/topics/helm-evaluation-platform#:~:text=The%20HELM%20platform%20conducts%20standardized%2C,The%20canonical%20workflow%20is). By structuring our simulator in this modular way, future extensions or changes (e.g., a new controller type, or a different evidence gating criterion) can be implemented by modifying or adding a module, without rewriting the entire orchestrator.

## 2. Unify Evidence Handling and Gating Logic

Evidence gating – ensuring that the student’s answer is supported by short, cited evidence – is currently implemented in multiple places (controllers, gating functions, evaluation checks), leading to duplicated logic and potential inconsistencies. We should create a single **evidence pipeline** with shared abstractions to handle everything from quote extraction to citation verification.

**Centralize quote processing:** Define a common utility for **quote tokenization and truncation**. All parts of the system should enforce the rule that quotes must be ≤ 15 tokens and ideally be verbatim substrings of the source text. Currently, this is handled piecemeal: - The Fact-Card extraction prompt instructs the model to use quotes ≤ 15 tokens, but we also see post-processing in orchestrator slicing quotes to 15 tokens if needed[[6]](file://file-D3gJ6oVMegQQNmR1sEttVb#:~:text=,toks%5B0%5D.start)[[7]](file://file-D3gJ6oVMegQQNmR1sEttVb#:~:text=slice_txt%20%3D%20opt%5Bstart_s%3Aend_e%5D%20c.setdefault%28,tokens%20slice). - The factcards.build\_option\_card\_ids function explicitly checks and only records a card if its quote length is ≤15 tokens[[8]](file://file-PAkVozqRNNKExGKLMyRB8r#:~:text=for%20oi%20in%20range%28len%28getattr%28task%2C%20,%3C%3D%2015). - The evidence signal computation filters out any cited card whose quote is longer than 15 tokens[[9]](file://file-P38QvTZmhYeiMHmqAinDVQ#:~:text=,import%20re%20as%20_re). These should be unified. We can create a helper like tokenize(text) -> List[str] (perhaps using a simple regex or an actual tokenizer) and truncate\_quote(text, max\_tokens=15) -> str. This way, whenever a quote is produced (during LEARN) or consumed (during evaluation), it passes through the same truncation logic. For example, after the student returns Fact-Cards, we can automatically clamp each quote to 15 tokens (to fix any model deviations) using one shared function, rather than repeating \_re.findall(...15...) in multiple places[[10]](file://file-D3gJ6oVMegQQNmR1sEttVb#:~:text=if%20len%28_re.findall%28r%22%5BA,toks%5B0%5D.start)[[11]](file://file-D3gJ6oVMegQQNmR1sEttVb#:~:text=end%20to%20the%2015th%20token,s%5D%20if%20toks).

**Shared evidence data model:** Use a common data structure for evidence across the system. The simulator already defines a Quote dataclass in evidence\_schema.py to represent a piece of evidence (fields: option letter, source\_id, text, start/end indices, card\_id)[[12]](file://file-Bn5VgZmZvLvkH6dC2teGdi#:~:text=%40dataclass%20class%20Quote%3A%20option%3A%20str,str%5D%20%3D%20None). We should adopt Quote as the standard representation after the LEARN phase. For instance, after extracting cards (which come in JSON with the schema shown in the handbook[[13]](file://file-9Fv1FZXe6jptdY185YjNN3#:~:text=Goal%3A%20extract%20discriminative%2C%20option%E2%80%91linked%20%E2%80%9Ccards%E2%80%9D,then%20cite%20them%20when%20answering)), convert them to Quote objects using adapt\_cards\_to\_quotes[[14]](file://file-Bn5VgZmZvLvkH6dC2teGdi#:~:text=def%20adapt_cards_to_quotes%28cards%3A%20List%5BDict%5Bstr%2C%20Any%5D%5D%20,append%28q%29%20return%20out). Then: - **Pre-gating**: Before the USE phase (answering), ensure that the set of extracted quotes covers all options adequately. Use a single function to check “coverage” per option. The current pass\_pre\_gates function does exactly this: it groups quotes by option and verifies each option has at least q\_min quotes from at least min\_sources[[15]](file://file-6LARtihGXyBfCZiDWt2cuf#:~:text=def%20pass_pre_gates%28quotes%3A%20List,min_sources%3A%20return%20False). This logic can be invoked right after LEARN. If pass\_pre\_gates fails (e.g., one option got 0 quotes), the FactCardManager (or orchestrator) can decide to trigger an escalation (another extraction round, if max\_learn\_boosts allows). By centralizing pre-gating here, all controllers or workflows benefit from the same check, rather than each implementing their own ad-hoc solution. The **fragile copy-paste** issue arises if, for example, a controller like Quote-then-Vote separately tried to ensure each option has a quote – by relying on one pass\_pre\_gates implementation, we avoid divergence. - **Post-gating and evaluation**: After the student gives an answer, we perform **witness and citation checks** using one pipeline. The pass\_post\_gates function in evidence\_gates.py encapsulates the main criteria: for the chosen option’s quotes, are there ≥ q\_min quotes, ≥ min\_sources distinct sources, and ≥ τ coverage overlap with the option text?[[16]](file://file-6LARtihGXyBfCZiDWt2cuf#:~:text=def%20pass_post_gates%28quotes%3A%20List,keys%28%29%29%2C%20chosen_letter) It returns both a boolean and a dict of evidence metrics (min quotes per option, chosen option coverage, witness overlap, etc.)[[17]](file://file-6LARtihGXyBfCZiDWt2cuf#:~:text=def%20evidence_health%28quotes%3A%20List,0%20srcs%20%3D%200)[[18]](file://file-6LARtihGXyBfCZiDWt2cuf#:~:text=,int%28srcs%29%2C). We should use this consistently to decide if the answer “passes” evidence requirements. Additionally, the **citation compliance** rules should be checked here: for example, **witness** requirement (does the chosen answer cite at least one card that quotes the chosen option text?) and **short quote** requirement (are all cited quotes ≤15 tokens and tagged with the current skill?). In the current code, these are checked in compute\_evidence\_signals – it filters cited cards by skill tag and quote length[[9]](file://file-P38QvTZmhYeiMHmqAinDVQ#:~:text=,import%20re%20as%20_re) and determines a has\_option\_quote map for each option[[19]](file://file-P38QvTZmhYeiMHmqAinDVQ#:~:text=%23%20has%20option,for%20c%20in%20cited_cards). We can incorporate those checks into our unified gating: - **Witness check**: Ensure has\_option\_quote[chosen\_index] is True for the answer’s citations (meaning at least one PRO card for that option was cited). This is essentially the “witness” rule (option-linked evidence)[[20]](file://file-9Fv1FZXe6jptdY185YjNN3#:~:text=LEARN%20rules%20,card%20quoting%20from%20a%20snippet)[[21]](file://file-9Fv1FZXe6jptdY185YjNN3#:~:text=USE%20%28answer%29%20JSON%20schema%20,...). - **Coverage and source count**: Ensure coverage ≥ τ and distinct source count ≥ min\_sources for the chosen option’s evidence. - **Citation tag compliance**: Ensure every cited card’s tags contains the current skill\_id and quote length ≤15, as required[[20]](file://file-9Fv1FZXe6jptdY185YjNN3#:~:text=LEARN%20rules%20,card%20quoting%20from%20a%20snippet). If any citation fails (e.g., the student tries to cite an unrelated card or a long quote), the answer should be marked as failing citation compliance. Our unified evidence pipeline can simply ignore or exclude any cards that don’t meet these criteria (as is done in compute\_evidence\_signals[[9]](file://file-P38QvTZmhYeiMHmqAinDVQ#:~:text=,import%20re%20as%20_re)) and use the filtered set for scoring. - **Scoring and feedback**: The evidence pipeline can compute a set of **evidence signals** to attach to the evaluation. Currently, compute\_evidence\_signals returns metrics like coverage, witness\_scores (how many token overlaps each option gets from the cited quotes), witness\_idx (which option has the highest overlap), and required\_ok (did the chosen option cite its required PRO card)[[22]](file://file-P38QvTZmhYeiMHmqAinDVQ#:~:text=,get%29%20if%20witness_scores%20else%20None)[[23]](file://file-P38QvTZmhYeiMHmqAinDVQ#:~:text=,letter%20in%20opt_map%20else%20None). These signals can be useful for analysis or even for reranking logic (the simulator uses witness\_scores in the evidence-based reranker to pick the answer with the highest supporting evidence[[24]](file://file-P38QvTZmhYeiMHmqAinDVQ#:~:text=,get%29%20if%20witness_scores%20else%20None)[[25]](file://file-JmJQjnemXAMnctRPSbj7DA#:~:text=def%20wscore%28i%3A%20int%29%20,1)). By computing these signals in one place, we avoid discrepancies. For instance, a controller shouldn’t need to separately calculate coverage – it can call the EvidenceGating module to get an updated coverage or a boolean witness\_pass.

**Eliminate copy-paste across controllers/gates/evaluators:** By centralizing all the above into an EvidenceGating class or module, we remove redundant code. The orchestrator (or controllers) no longer have to duplicate checks. For example, instead of each evaluator or controller checking “if no option quote, maybe penalize or escalate”, they all rely on pass\_pre\_gates and pass\_post\_gates. We saw that evidence\_gates.py and evidence.py had overlapping functionality: both define token extraction and coverage calculation in slightly different ways (one on Quote objects, one on raw card JSON)[[26]](file://file-6LARtihGXyBfCZiDWt2cuf#:~:text=def%20coverage_for_option%28quotes%3A%20List,ot)[[22]](file://file-P38QvTZmhYeiMHmqAinDVQ#:~:text=,get%29%20if%20witness_scores%20else%20None). Merging these into one pipeline ensures that the **same definition of coverage** and **the same tokenization logic** is used everywhere. It’s also easier to update in the future (e.g., if we refine the coverage metric or add a new criterion like requiring two independent sources).

**Reference to project modules:** The importance of this unification is underscored by comments in our code. The evidence helper module explicitly notes it was created to reduce clutter in orchestrator and enable unit testing[[27]](file://file-P38QvTZmhYeiMHmqAinDVQ#:~:text=This%20module%20isolates%20small%2C%20testable,). We should extend that principle: keep all evidence-related logic in sim.evidence (or similar), and let orchestrator/controllers just call it. The **Fact-Card Handbook** (v2) details the evidence rules[[20]](file://file-9Fv1FZXe6jptdY185YjNN3#:~:text=LEARN%20rules%20,card%20quoting%20from%20a%20snippet)[[28]](file://file-9Fv1FZXe6jptdY185YjNN3#:~:text=text%20via%20%60where.scope%3D,card%20quoting%20from%20a%20snippet) – our single pipeline should be the authoritative implementation of those rules (ensuring a cited verbatim quote for the correct option, adding con quotes for distractors, etc., though generation of those is on the tutor/student side).

In summary, **unifying evidence logic** means one module handles: 1. Converting raw cards to Quote objects and clamping quotes to ≤15 tokens. 2. Pre-answer gating (each option has enough evidence). 3. Post-answer gating (chosen option evidence sufficiency and correctness of citations). 4. Calculating evidence metrics (coverage, overlaps, source counts, etc.). 5. Final credit decision: determine if the student gets credit for a correct answer. For example, in strict citation mode, a correct answer is only “credited” if evidence conditions are satisfied[[29]](file://file-9Fv1FZXe6jptdY185YjNN3#:~:text=Record%20shape%20%28MCQ%29%20%60%60%60%20,432%2C). The unified gating check can set a flag like credited=True/False accordingly.

By doing so, all parts of the simulator will “speak the same language” regarding evidence, and any future adjustments (like a different coverage threshold or a new metric like evidence novelty) can be made in one place.

## 3. De-Duplicate Utility Functions and Constants

During the refactor, we should gather all the small utility functions and constants scattered across modules and consolidate them in a shared utilities module (e.g., sim.utils or sim.shared). Currently, many such helpers are duplicated or inconsistently used:

* **Tokenization and text processing**: As noted, we have multiple regex tokenizers (e.g., \_TOK = re.compile(r"[A-Za-z0-9]+") in evidence\_gates.py[[30]](file://file-6LARtihGXyBfCZiDWt2cuf#:~:text=_TOK%20%3D%20re.compile%28r%22%5BA), similar logic in card\_quality.tokens, and ad-hoc \_re.findall in orchestrator). Create a single tokens(text: str) -> List[str] utility (perhaps in sim.utils.text) that returns token list or count. All modules should use this for counting tokens (for quote length enforcement, overlap computation, etc.), to avoid subtle differences (e.g., one might treat hyphenated words differently than another if regex differs).
* **Quote clamp and slicing**: Introduce first\_n\_tokens(text, n) and/or first\_n\_tokens\_span(text, n) in utils, which return the first *n* tokens of a text (and optionally the character span if needed). The orchestrator currently uses a function first\_n\_tokens\_span from sim.validation to handle fallback slicing[[31]](file://file-D3gJ6oVMegQQNmR1sEttVb#:~:text=c%5B,context) and even adjusts span boundaries to exactly n tokens[[32]](file://file-D3gJ6oVMegQQNmR1sEttVb#:~:text=,)[[11]](file://file-D3gJ6oVMegQQNmR1sEttVb#:~:text=end%20to%20the%2015th%20token,s%5D%20if%20toks). These low-level details can be encapsulated in a well-documented helper. For example:
* def clamp\_quote\_to\_n(text: str, n: int) -> str:  
   tokens = tokenize(text)  
   return " ".join(tokens[:n])
* This simple approach may suffice for truncation. If we need to preserve original context spans (for traceability of where the quote came from), a first\_n\_tokens\_span can return both the truncated text and the new end index. In any case, having one implementation avoids divergence (currently, the orchestrator’s span logic[[11]](file://file-D3gJ6oVMegQQNmR1sEttVb#:~:text=end%20to%20the%2015th%20token,s%5D%20if%20toks) and factcards.build\_option\_card\_ids logic[[8]](file://file-PAkVozqRNNKExGKLMyRB8r#:~:text=for%20oi%20in%20range%28len%28getattr%28task%2C%20,%3C%3D%2015) both handle the 15-token clamp).
* **Option mapping (index ↔ letter)**: The simulator often maps option indices (0,1,2,…) to letters (A, B, C, …) and vice versa. We see this in multiple places: constructing choice strings for outputs, assigning card IDs per option (option\_card\_ids maps like {"A": "f1", ...} in FactCard processing[[33]](file://file-PAkVozqRNNKExGKLMyRB8r#:~:text=def%20build_option_card_ids%28new_cards%3A%20List,first%20PRO%20card%20per%20option)[[8]](file://file-PAkVozqRNNKExGKLMyRB8r#:~:text=for%20oi%20in%20range%28len%28getattr%28task%2C%20,%3C%3D%2015)), and in evidence checks (e.g., chr(ord('A')+i) in evidence signal calc[[23]](file://file-P38QvTZmhYeiMHmqAinDVQ#:~:text=,letter%20in%20opt_map%20else%20None)). Create a small utility for this, e.g.:
* def index\_to\_letter(i: int) -> str: # 0 -> "A"  
  def letter\_to\_index(letter: str) -> int:
* By using these consistently, we avoid off-by-one or casing issues. It also centralizes the assumption of at most 26 options (A-Z); if in future tasks with more options appear, this utility can be updated (e.g., AA for 27, etc.) in one place.
* **Card filtering and deduplication**: There are functions like factcards.trim\_card (drop unused fields in a card dict) and factcards.dedup\_per\_option (remove near-duplicate quotes per option using Jaccard similarity over tokens)[[34]](file://file-PAkVozqRNNKExGKLMyRB8r#:~:text=def%20dedup_per_option%28cards%3A%20List%5BDict%5Bstr%2C%20Any%5D%5D%2C%20,scoped%20cards%20while%20preserving%20order)[[35]](file://file-PAkVozqRNNKExGKLMyRB8r#:~:text=by_opt%3A%20Dict,float). These are general utility behaviors (trimming to needed schema, removing duplicates) that could live in a sim.utils.cards module or within the FactCardManager. They use common constants such as dedup\_sim threshold (default 0.88)[[36]](file://file-RTVkGhp9td6zcwN9SkVRfD#:~:text=window%20around%20hits%20for%20retrieval%2Fcontext,len%22%2C%20dest%3D%22min_card_len%22%2C%20type%3Dint%2C%20default%3D40) and length limits (min\_card\_len, max\_card\_len). We should **standardize expectations for card content**:
* Document that quotes should be between X and Y characters (configurable via min\_card\_len/max\_card\_len).
* Document that if a quote is outside these bounds, it may be filtered out or truncated.
* Ensure one place in code applies these filters. For example, after extraction, we can filter out cards with quote length < 40 chars or > 300 chars (per config) to remove trivial or overlong cards, using one function referencing cfg.dials.min\_card\_len etc. This prevents different parts accidentally using different thresholds.
* Same for **Card Quality Score (CQS)**: The simulator likely computes a quality score for each card (perhaps based on relevance or uniqueness). If so, enforce min\_cqs in one spot (e.g., the FactCardManager filtering step). The CLI and config mention a recommended min\_cqs=0.55 when enabled[[37]](file://file-RTVkGhp9td6zcwN9SkVRfD#:~:text=help%3D%22near,k%22%2C%20dest%3D%22per_option_top_k%22%2C%20type%3Dint); such domain-specific constants should be applied uniformly. A utility function compute\_cqs(card) could live in sim.card\_quality so that both the selection of top quotes and the evidence-weighted option ranking use the same values.
* **Miscellaneous**: Other duplicated patterns include retrieving environment variables for config (the CLI does this for a few toggles; a util could standardize env override handling), and mapping from config flags to internal behavior (e.g., grammar modes "none", "json", "schema" – a utility can map these strings to actual decoding parameters or schema files).

By moving these into sim.utils (or perhaps grouping by category: sim.utils.text, sim.utils.cards, etc.), we reduce clutter in the main modules and ensure consistency. For instance, if sim.utils.text.tokens is the single source of truth for tokenization, then evidence\_gates.coverage\_for\_option and card\_quality.jaccard can both rely on it, producing consistent results. When developers update how tokens are defined (say to handle Unicode letters or punctuation differently), they update one place.

**Document the standards**: It’s important to clearly **document standard expectations** for evidence quotes and related utilities: - *Quote length:* “Quotes should contain at most 15 tokens (per the Fact-Card schema and evaluation rules)[[38]](file://file-9Fv1FZXe6jptdY185YjNN3#:~:text=ICL_Simulator_Handbook,then%20cite%20them%20when%20answering)[[20]](file://file-9Fv1FZXe6jptdY185YjNN3#:~:text=LEARN%20rules%20,card%20quoting%20from%20a%20snippet). The simulator will automatically truncate any longer quote to 15 tokens.” - *Quote scope:* “Quotes must be verbatim substrings of either an option text (scope="option") or a context snippet (scope="context"). Option-scoped quotes are required for the correct answer (to serve as a ‘witness’). The system will verify and adjust quotes to ensure they are exact matches in the source. If a returned quote is not found in the supposed source text, the simulator may replace it with a leading substring of the source (to enforce consistency).” - *Tagging:* “Every Fact-Card must include the current skill’s tag in its tags list[[39]](file://file-D3gJ6oVMegQQNmR1sEttVb#:~:text=,or). The simulator adds the tag if missing. This is used to ensure that cards from previous skills or irrelevant info are not credited during evaluation.” - *Option labeling:* “Options are labeled with capital letters (A, B, C, ...). These are used in the JSON outputs and for mapping citations to options. Utility functions handle this mapping to avoid mistakes.” - *Duplicate quotes:* “The simulator filters out near-duplicate quotes for the same option (default Jaccard similarity ≥ 0.88 considered duplicate). Only the first unique quote per option is kept, up to per\_option\_top\_k quotes[[40]](file://file-RTVkGhp9td6zcwN9SkVRfD#:~:text=p.add_argument%28%22,citations%22%2C%20default%3DNone%2C%20choices%3D%5BNone%2C%20%22off%22%2C%20%22lenient). This ensures diversity in evidence and that the model doesn’t just repeat the same snippet.”

Having these conventions in one place in the code (as constants or clearly named functions) and in documentation makes the behavior predictable. It also helps anyone extending the system (e.g., adding a new retrieval tool) to know how their contributions will be processed (for example, if they add 5 snippets per option, the dedup step might drop some – they should know the criteria).

In summary, consolidating utilities in sim.utils removes redundancy and ensures **one authoritative implementation** for each small operation. It will reduce bugs (since fixes apply globally) and make the codebase easier to understand (since utils functions can carry descriptive names, replacing inline regex logic).

## 4. Simplify and Validate Configuration & CLI Usage

Over time, sim.config and sim.cli have accumulated many options and flags, some core to functionality and others experimental or legacy. We should restructure the configuration into clearer layers, prune or isolate rarely used knobs, and provide easier ways to manage configurations (like presets or config files).

**Layer the configuration “dials”:** - **Core dials:** These are options that most directly affect the simulator’s primary behavior. They should be easy to discover and tweak. For example: closed\_book, anonymize, reasoning mode (none vs CoT vs others), self\_consistency\_n, use\_fact\_cards, require\_citations, controller type, etc. These define major evaluation dimensions and should be prominent. In code, we might group these in the CLI help and documentation as “Core Settings”. - **Secondary/experimental knobs:** Options that are useful for research experiments or fine-tuning but not essential for every user. This includes things like adaptive\_sc (early stopping self-consistency), sc\_quorum, the various ToT parameters (tot\_width, tot\_depth, tot\_judge), entropy gating flags (uncertainty\_gate, thresholds), and the few-shot selection parameters (embedding backend choice, MMR lambda, etc.). These can be grouped under an “Advanced” section in CLI or even require an explicit flag like --advanced to show them, to avoid overwhelming new users. Internally, they can still live in the Dials dataclass, but clearly marked (via comments or naming) as advanced. - **Retired or debug flags:** Identify flags that are no longer yielding significant benefits or were for debugging purposes. For example, reflection\_every is noted as “not implemented yet”[[41]](file://file-6deT6fS8QekMe65HnEcU21#:~:text=anonymize%3A%20bool%20%3D%20True%20rich%3A,0%3Dauto%20majority%20of) – unless we plan to implement it soon, we can remove it or hide it. The rare\_emphasis flag is a placeholder that might not do anything substantive yet. Flags like --no-anon vs --anonymize might be simplified (the CLI already treats anonymization as on by default and uses --no-anon to disable). We should also consider deprecating any environment-variable-based toggles in favor of explicit config (environment overrides can still exist, but primary usage should be via config/CLI for clarity).

**Validate combinations and provide presets:** Because there are many interdependent options, we should build a validation layer that checks for incompatible or nonsensical combinations. For example: - If dials.grammar = "json" but the model does not support constrained decoding, maybe warn the user. - If self\_consistency\_n > 1 and controller = "tot", ensure the controller itself can handle multiple votes or clarify how they interact (perhaps ToT inherently uses multiple samples). - If use\_fact\_cards=False but require\_citations=True, the expectation is that citations will come from tools or be impossible – the system should warn if no mechanism to produce citations is enabled (since closed-book with no retrieval and no fact-cards would make require\_citations impossible to satisfy). - If controller="ltm" (Let’s think step-by-step) and reasoning="tot" (Tree of Thoughts) both set, is that valid or conflicting? Possibly the reasoning flag overlaps with controller choices (the CLI options like --reasoning tot vs --controller tot might confuse users). We should clarify that --controller selects a multi-step orchestration, whereas --reasoning selects a prompting style (e.g., cot adds a “Let’s think…” header). In validation, we might ensure that incompatible or redundant combinations are resolved (perhaps if controller is not basic, ignore the reasoning prompt style beyond the first step).

One approach to managing complexity is to introduce **named presets or configurations**. Using frameworks like **Hydra** or **Sacred** can help manage hierarchical configs and even allow command-line overrides. For example, we could define YAML preset files for common scenarios: - closed\_book\_strict.yaml – closed-book, fact-cards enabled, strict citation gating, maybe self\_consistency=1 (single shot). - open\_book\_baseline.yaml – (if open-book mode existed or to simulate an oracle). - fast\_debug.yaml – perhaps uses a mocked tutor and fewer steps for quick testing. - tot\_strategy.yaml – turns on Tree-of-Thought controller with certain width/depth and gating.

Each preset would populate a RunConfig/Dials with certain values. The user can then do sim.cli --config closed\_book\_strict.yaml or similar. Even without introducing an external library, we can simulate this by providing a few example config files and a small loader. This also ties into experiment tracking: each preset or combination can be given an **experiment name or ID**.

For instance, in the example CSV, we see model\_slug or model\_family fields like mixtral\_sc3\_b6\_confirm which likely encode a particular mix of settings (maybe SC=3, best\_of=6, etc.). Instead of encoding in the name manually, we could allow tagging a run with a run\_name or preset. The CLI can have a --preset <name> which loads known settings (essentially a mini Hydra). This makes it easier to compare runs – if one run was using --preset strict\_cot and another --preset relaxed\_no\_citations, the output logs or results can carry those labels.

**CLI ergonomics:** The CLI argument list is quite long (as we saw, dozens of flags)[[42]](file://file-RTVkGhp9td6zcwN9SkVRfD#:~:text=def%20main%28argv%3A%20list%5Bstr%5D%20,task%20type%20to%20run)[[43]](file://file-RTVkGhp9td6zcwN9SkVRfD#:~:text=%23%20Controllers%20p.add_argument%28%22,currently%20%27self%27%20only). To simplify: - Use argument **groups** in argparse to organize help output (e.g., “Evidence & Gating Options”, “Few-Shot Options”, “Controllers”, etc.). This doesn’t change functionality but helps users navigate. - Provide sane defaults and maybe **short aliases** for frequently used flags. For example, -N for --num\_steps, -k for --shots-k, etc., if it makes sense. - Consider **removing some redundancy**: The CLI currently has both --fact-cards (bool) and --use\_fact\_cards in config, or --require-citations vs --citations strict/lenient. We might unify these: e.g., just use --citations off|lenient|strict with no separate boolean. If strict is requested, it implies require\_citations. The code in CLI already does some of this logic[[44]](file://file-RTVkGhp9td6zcwN9SkVRfD#:~:text=help%3D,idk%22%2C%20dest%3D%22idk_enabled%22%2C%20action%3D%22store_true)[[45]](file://file-RTVkGhp9td6zcwN9SkVRfD#:~:text=,on), but we can expose it more cleanly to the user. - Ensure the CLI help text clearly states interactions (for example, help for --require-citations could note “implies --citations strict if not otherwise set”).

**Example of a layered config (conceptual):** Using Hydra-like syntax for illustration:

defaults:  
 - preset: closed\_book\_strict  
  
preset:  
 closed\_book\_strict:  
 dials:  
 closed\_book: true  
 use\_fact\_cards: true  
 citation\_mode: strict  
 self\_consistency\_n: 1  
 controller: basic  
 require\_citations: true  
 fast\_debug:  
 dials:  
 closed\_book: true  
 use\_fact\_cards: false  
 self\_consistency\_n: 1  
 controller: basic  
 num\_steps: 2  
 skill\_id: "debug\_skill"

The idea is that a user or experiment script can switch preset to quickly toggle a bundle of settings. Internally, we can implement this without Hydra by having a dictionary of presets mapping to RunConfig values.

**Comparison and Reproducibility:** Named configs or presets also facilitate the **top-level comparisons** the user might do. For instance, running scripts/experiment.py with different presets and comparing results can be automated if each run knows its config name. We see the system prints a run\_id and possibly uses the config to name output files (like the JSONL filename contains mixtral\_sc3\_b6\_confirm). We should formalize that: - Perhaps generate a short hash or name based on critical config values (like a digest of the Dials) to tag the run. The run\_id (UUID) is great for uniqueness, but a human-readable label helps when aggregating. We could include cfg.alias\_family\_id or a new field experiment\_name in RunConfig to carry this. - Encourage using these tags in filenames or in the CSV summary. For example, having a column for preset or config\_name in the summary CSV would allow grouping results easily.

**Hydra/Sacred inspiration:** If adoption of a library is feasible, Hydra offers structured config composition and command-line overrides (e.g., python sim.py preset=closed\_book\_strict dials.sc\_policy=adaptive). Sacred provides experiment tracking and configuration logging. At minimum, our design can mimic some of their benefits: layering defaults and overrides, and printing the full config for each run (so results are reproducible). Ensuring reproducibility also means capturing all random seeds and relevant parameters – we already pass anon\_seed for anonymization reproducibility[[46]](file://file-9Fv1FZXe6jptdY185YjNN3#:~:text=,for%20reproducible%20runs), and we might do similar for any sampling randomness.

**Simplify config code:** The RunConfig and Dials dataclasses are a good structure (type-checked, with defaults)[[47]](file://file-6deT6fS8QekMe65HnEcU21#:~:text=%40dataclass%20class%20Dials%3A%20,pre%7Cpost%7Cnone)[[48]](file://file-6deT6fS8QekMe65HnEcU21#:~:text=,strict%20%28strict%20enforces%20hard%20gates). We should: - Remove any fields that are no longer used (to avoid confusion). - Potentially split Dials into sections (maybe logically, not necessarily as separate classes): e.g., RetrievalDials, EvidenceDials, etc., to organize them. However, this might complicate usage. An alternative is just to organize the declaration with comments (as is done) and keep it flat for simplicity. - The CLI parsing should map directly into RunConfig without too many tweaks. Right now, CLI does some post-processing (like computing citation\_mode from flags and env)[[45]](file://file-RTVkGhp9td6zcwN9SkVRfD#:~:text=,on)[[49]](file://file-RTVkGhp9td6zcwN9SkVRfD#:~:text=citation_mode%20%3D%20None%20if%20args,strict). This can be encapsulated in a function or within Dials (e.g., a method that takes args and environment and returns a ready Dials). That would make cli.py shorter and less error-prone.

In short, the configuration should be made **clear, layered, and user-friendly**. By doing so, users can confidently adjust high-level settings without wading through extraneous options, and advanced users can still find the fine-tuning knobs when needed. Clean config management also reduces the chance of conflicting settings and makes experiments easier to replicate or share.

## 5. Improve Logging and Analysis Pipeline

Robust logging is key for analyzing within-session learning and forgetting. The simulator already produces JSONL logs per step and has a script to aggregate them into CSV[[29]](file://file-9Fv1FZXe6jptdY185YjNN3#:~:text=Record%20shape%20%28MCQ%29%20%60%60%60%20,432%2C). We propose to further normalize and enrich these logs to facilitate analysis.

**Consistent record format:** Each step’s JSON record should have a consistent schema, regardless of which features are enabled. This means even if certain fields are not applicable, they should appear (perhaps as null or default). For example: - Always include an "evaluation" section with "correct": bool. If citations/evidence gating are in play, include a subfield for "citations\_evidence" with keys like "coverage", "witness\_pass", "credited", etc. In a run where no evidence gating is used, we can still output "citations\_evidence": {"coverage": 0.0, "witness\_pass": true, "credited": true} or simply an empty object, to keep the structure consistent. (Alternatively, omit the subobject entirely if not applicable, but then the CSV columns might misalign.) - Always log the chosen answer and any auxiliary info. For MCQ, we have "answer": {"chosen\_index": X, "votes": [...], "citations": [...]}[[29]](file://file-9Fv1FZXe6jptdY185YjNN3#:~:text=Record%20shape%20%28MCQ%29%20%60%60%60%20,432%2C). For an SAQ or free response, "answer": {"response": "...", "citations": [...]}" could be the format (ensuring a citations list is present if evidence was required). - Include fields for **timing and usage** uniformly: e.g., "duration\_ms" (time taken for that step), "student\_usage" (token counts, API calls)[[50]](file://file-9Fv1FZXe6jptdY185YjNN3#:~:text=,432%2C). If any controllers incur multiple calls, ensure those are summed up or logged in a clear way (the example CSV tracks student\_calls, student\_total\_tokens, etc. per step).

**Normalize across runs:** The CSV aggregator already merges multiple runs. To make that reliable, each JSON log line should have metadata identifying the run and step. We might include in each JSON record a "run\_id" and "step\_index", so that if multiple logs are concatenated the context isn’t lost. The example CSV appears to be the output of a script combining runs (with columns for run\_id, model\_name, etc.). We can improve by: - Logging a **header record** at the start or as separate metadata, containing the fixed config for that run (skill, task type, difficulty, all dials used). This could be a JSON object with "run\_config": { ... }. That way the log file self-documents the conditions. The analysis script can copy those into each row of the CSV for convenience (as is done in the example, where many columns repeat the config and model info). - Ensuring every step log has the same fields, so the CSV columns remain consistent. If a field is missing in some records, the CSV might become ragged or require special handling. For example, if a run never uses tools, maybe "tools\_used" is empty – better to still log "tools\_used": [] or such.

**Single CSV export for metadata:** The aim is that all important metadata across steps and runs can be compiled into a single table for analysis. The provided steps.csv illustrates this: each row has dozens of columns capturing config, step outcome, evidence metrics, timing, etc.. We should ensure that for every new metric or piece of data we log, a column will appear. Some suggestions: - Make sure **evidence metrics** like coverage are logged as numeric values, even if evidence gating is off (could be default 0 or NaN). In the example, we see coverage 1.0 or 0.0, witness\_pass booleans, etc. being logged for each step. This allows filtering for cases where, say, coverage was below τ to analyze failures. - Log **failure reasons or flags** explicitly. The example CSV has failure\_reasons like no\_option\_quote\_for\_choice;coverage\_below\_tau;witness\_mismatch for steps that were answered incorrectly or not credited. We can standardize a set of reason codes (e.g., "no\_option\_quote", "coverage\_below\_tau", "witness\_mismatch", "citation\_missing" etc.) that the EvidenceGating module produces when a requirement is not met. Logging these as a list (or semicolon-separated string) is useful for debugging why a student failed to get credit. - If **abstention/IDK** is enabled, log whether the student abstained on that step and why. The CSV has abstained and abstain\_reason columns. Ensure these are present even when IDK is off (with abstained=false by default) so the schema stays fixed. - **Tags for analysis**: We might include a column or field for any special experimental condition, e.g., an alias swap ID or a difficulty tag. In the future, if multiple skills per session are allowed, we’d log the skill for each step (currently skill\_id is probably fixed per run; but if it varied, we’d include it per step). In the CSV, skill\_id is present but blank in the snippet, perhaps because that experiment didn’t fix a single skill (or it wasn’t logged properly). We should ensure skill\_id (or some identifier of the content being tested) is always logged per step.

**Comparison by experiment ID:** We already generate a run\_id (likely a UUID) and have fields like model\_name or model\_family to identify which model was used in the run. To facilitate comparisons: - Encourage including a **run descriptor** in the JSON/CSV. For example, if we have presets or an alias, include a config\_name or experiment\_id that is human-meaningful (e.g., “mixtral\_sc3\_b6\_confirm” as in the CSV). This can be set via CLI or automatically constructed. It helps group runs when plotting learning curves, etc. - The analysis script can use this field to pivot results. For example, comparing average accuracy over steps for runs with with\_citations vs without\_citations presets.

**Traceability and Reproducibility:** Logging should also serve to reproduce results if needed. This means recording: - The exact prompts used (perhaps in a debug log, not necessarily in the main JSON). - The random seeds (we see anon\_seed in environment for anonymization[[46]](file://file-9Fv1FZXe6jptdY185YjNN3#:~:text=,for%20reproducible%20runs); similarly, if there’s any nondeterminism in student sampling beyond what’s in the config, log the seed). - Version of the simulator or git commit if available (to track code changes between runs). - The tutor model version (the handbook mentions a fixed tutor gpt-5-nano-2025-08-07[[51]](file://file-9Fv1FZXe6jptdY185YjNN3#:~:text=,and%20writes%20a%20summary%20CSV); if that changes, log it). - Any external data references (if using a retrieval tool, maybe log which corpus or index it was querying, if applicable).

Most of these might be overkill for the step log, but could be included in the run header metadata.

**Making analysis easier:** With logs normalized, the scripts/analyze.py can read JSONL or the aggregated CSV and compute metrics like: - Accuracy over time (does correct improve from step 1 to N?). - Evidence usage over time (does coverage or quote count improve, indicating the student learning to gather evidence?). - Breakdown by conditions (e.g., compare runs with/without fact-cards). Thanks to consistent fields, these analyses can be done with simple grouping and filtering. For example, one could filter credited==false and correct==true to find all cases where the student knew the answer but failed to cite evidence – a measure of within-session learning without showing work, perhaps.

Finally, encourage using the logged **experiment tags** or IDs for comparison. For example, if each run has an experiment label, one can join or merge multiple runs’ CSVs and use that label to distinguish them. It’s also useful for visualization (plot lines labeled by experiment). The user should be advised to always either use --log to capture output and then run the provided analysis script which outputs such a CSV, or to use the integrated logging in a Jupyter environment for quick analysis.

By improving logging in these ways, the simulator becomes not just a tool for generating data, but a tool for **measuring** and **diagnosing** in-context learning behavior. The emphasis should be on making all relevant data accessible in one place (the CSV), so that adding a new metric or conducting a new analysis doesn’t require rerunning or manual log parsing.

## 6. Design for Future Extensions

Lastly, while refactoring we should keep an eye on future needs, ensuring the architecture is flexible for anticipated complexity. Some forward-compatible design points include:

* **Streaming or Multi-turn Students**: In future, the simulator may include interactive students or ones that learn over multiple turns of dialogue (for example, a student that asks questions back, or a tutor-student multi-turn exchange). Our orchestrator should accommodate this by not assuming each task is a single question/answer. Perhaps define an abstract interface for a “Lesson” or “Session” that could contain multiple interactions. We already have a StatefulLLMStudent stub (for --student stateful-llm) to maintain state across turns[[52]](file://file-RTVkGhp9td6zcwN9SkVRfD#:~:text=if%20args.student%20%3D%3D%20,if%20args.notes_file). To support this, we might:
* Structure the FactCardManager or context assembly to be updated incrementally after each turn.
* Allow controllers to loop until some condition or number of turns is reached.
* Ensure the logging can record multi-turn exchanges (e.g., step 1a, 1b for Q and A).

By decoupling the concept of a “step” (which might currently equate to one question-answer pair) from the logging and control flow, we can handle sub-turns. For example, treat each QA pair as a sub-step but still under the umbrella of one learning step. The orchestrator could call learner.answer() multiple times per step if needed. Designing the controllers as pluggable already helps – a future “DialogueController” could implement a loop of interactions for a single high-level task.

* **Hard-negative Retrieval for Fact-Cards**: We might want to challenge the student with retrieved evidence that includes *distractors* (irrelevant or confounding snippets). This could enhance the Fact-Card extraction by forcing the model to discern which snippets are actually useful. To prepare for this:
* The retrieval tool interface can be designed to return snippets with labels or scores, so that later we can mark some as “hard negatives”. The FactCardManager could then potentially label cards coming from hard negatives differently (maybe they would be expected to be cited as con cards rather than pro).
* When computing evidence signals, we might incorporate whether a cited source was a known distractor. For now, just ensure that the evidence pipeline and data structures can carry a “source type” or some metadata. For example, extend Quote or cards with a field indicating if it’s a snippet vs direct option text vs hard-negative. This allows future gating like “don’t credit if only negative sources were cited” or similar.
* The orchestrator could allow a mode where after initial Fact-Card extraction, an extra step injects a misleading snippet and sees if the student updates their answer. This could reuse the modular structure (perhaps a specialized controller or a flag in FactCardManager to include negatives). By having clear separation of retrieval, we can slot this in later.
* **Multiple skills learning per session**: The current simulator typically focuses on one skill (knowledge area) per run, to measure within-session improvement on that skill. Future experiments might involve **curricula** or interference between skills – e.g., learning skill A, then skill B, then testing retention of A. To be ready for this:
* Don’t hard-code a single skill\_id throughout a run. While RunConfig.skill\_id is optional[[53]](file://file-6deT6fS8QekMe65HnEcU21#:~:text=,general), we could allow it to be a list or sequence in the future. Our data structures (Fact cards, quotes) already tag each card with a skill tag[[54]](file://file-9Fv1FZXe6jptdY185YjNN3#:~:text=12%7D%2C%20,%5D%7D). We should ensure the FactCardManager can handle cards from multiple skills coexisting, perhaps distinguishing them to avoid inappropriate reuse of cards when the question changes topic.
* The orchestrator loop could be extended to iterate over a list of skills, or even dynamically decide to switch skill mid-run. Our logging should then record the skill per step (which we plan to do). Also, the evidence gating must ensure that when a new skill starts, old cards either are cleared or at least not counted for evidence (unless we simulate a student who can use previously learned cards on new topics, which is an interesting scenario but then the evidence gating should check the tag and catch that as a violation if used across skills).
* Possibly introduce a concept of “session memory” vs “working memory”. Maybe FactCardManager could one day allow only same-skill cards in the current context for strict closed-book, but keep an archive of others to measure forgetting. While this is beyond current scope, our design of FactCardManager and EvidenceGating with skill tags will make sure the system is aware of which skill’s evidence is being used. Indeed, we already filter cited cards by current skill tag when computing evidence signals[[55]](file://file-P38QvTZmhYeiMHmqAinDVQ#:~:text=evidence) to prevent cross-topic credit.
* **Scalability and performance**: As complexity grows (multi-turn, more evidence processing), the simulator should remain fast. This means our refactored components should avoid unnecessary overhead. For instance, converting cards to Quote objects each step is fine (they are few), but if we had hundreds of cards, we might want to cache token sets for overlap calculation. We can foresee possibly parallelizing some parts (like self-consistency calls) or caching embeddings for example selection. While not explicitly asked, a modular design will naturally allow swapping in optimized versions (e.g., a faster evidence overlap function in C, or using vectorized operations for coverage). By keeping functions pure and focused (e.g., coverage\_for\_option(quotes, text)[[26]](file://file-6LARtihGXyBfCZiDWt2cuf#:~:text=def%20coverage_for_option%28quotes%3A%20List,ot) could be optimized internally without changing its interface), we enable performance tuning later.

In conclusion, anticipating these future requirements validates the refactoring approach: - **Modularity** ensures that adding a DialogueController or a multi-skill curriculum scheduler doesn’t break the existing flow – they can plug into the orchestrator framework. - **Unified evidence logic** with skill tagging already guards against multi-skill leakage and can be extended to handle more complex evidence scenarios. - **Clean config and logging** will be crucial as experiments diversify (you might have config flags like --curriculum or --negative-snippets later; having a structured config makes adding those straightforward, and logging ensures you can trace their effects). - **Extensibility** is explicitly a goal of HELM and similar platforms[[56]](https://www.emergentmind.com/topics/helm-evaluation-platform#:~:text=), and our simulator aims to be a general-purpose ICL sandbox, so we should follow those principles now to avoid technical debt later.

By implementing the above refactorings – untangling the orchestrator, unifying evidence handling, consolidating utilities, streamlining configuration, improving logging, and designing with future expansions in mind – we will transform the ICL simulator into a cleaner, more maintainable system. This not only makes the current goals achievable (measuring within-session learning fairly with strict JSON outputs and evidence gating) but also poises the simulator to incorporate new ideas and research directions with minimal friction. Each component will have a clear responsibility, mirroring the modular, extensible design of successful evaluation frameworks[[57]](https://www.emergentmind.com/topics/helm-evaluation-platform#:~:text=Toolkit%20implementation%3A)[[58]](https://www.emergentmind.com/topics/helm-evaluation-platform#:~:text=,world%20AI%20research), and the whole system will be easier to test, debug, and evolve.

[[1]](https://www.emergentmind.com/topics/helm-evaluation-platform#:~:text=Toolkit%20implementation%3A) [[5]](https://www.emergentmind.com/topics/helm-evaluation-platform#:~:text=The%20HELM%20platform%20conducts%20standardized%2C,The%20canonical%20workflow%20is) [[56]](https://www.emergentmind.com/topics/helm-evaluation-platform#:~:text=) [[57]](https://www.emergentmind.com/topics/helm-evaluation-platform#:~:text=Toolkit%20implementation%3A) [[58]](https://www.emergentmind.com/topics/helm-evaluation-platform#:~:text=,world%20AI%20research) HELM Evaluation Platform

<https://www.emergentmind.com/topics/helm-evaluation-platform>

[[2]](https://www.emergentmind.com/topics/big-bench-benchmark#:~:text=3) BIG-Bench Benchmark

<https://www.emergentmind.com/topics/big-bench-benchmark>

[[3]](file://file-6deT6fS8QekMe65HnEcU21#:~:text=This%20module%20hosts%20small%2C%20tidy,exports%20these%29.) [[41]](file://file-6deT6fS8QekMe65HnEcU21#:~:text=anonymize%3A%20bool%20%3D%20True%20rich%3A,0%3Dauto%20majority%20of) [[47]](file://file-6deT6fS8QekMe65HnEcU21#:~:text=%40dataclass%20class%20Dials%3A%20,pre%7Cpost%7Cnone) [[48]](file://file-6deT6fS8QekMe65HnEcU21#:~:text=,strict%20%28strict%20enforces%20hard%20gates) [[53]](file://file-6deT6fS8QekMe65HnEcU21#:~:text=,general) config.py

<file://file-6deT6fS8QekMe65HnEcU21>

[[4]](file://file-JmJQjnemXAMnctRPSbj7DA#:~:text=def%20apply_best_of_rerank%28%20,str%2C%20Any%5D%5D%2C%20notes_buf%3A%20str) [[25]](file://file-JmJQjnemXAMnctRPSbj7DA#:~:text=def%20wscore%28i%3A%20int%29%20,1) rerank.py

<file://file-JmJQjnemXAMnctRPSbj7DA>

[[6]](file://file-D3gJ6oVMegQQNmR1sEttVb#:~:text=,toks%5B0%5D.start) [[7]](file://file-D3gJ6oVMegQQNmR1sEttVb#:~:text=slice_txt%20%3D%20opt%5Bstart_s%3Aend_e%5D%20c.setdefault%28,tokens%20slice) [[10]](file://file-D3gJ6oVMegQQNmR1sEttVb#:~:text=if%20len%28_re.findall%28r%22%5BA,toks%5B0%5D.start) [[11]](file://file-D3gJ6oVMegQQNmR1sEttVb#:~:text=end%20to%20the%2015th%20token,s%5D%20if%20toks) [[31]](file://file-D3gJ6oVMegQQNmR1sEttVb#:~:text=c%5B,context) [[32]](file://file-D3gJ6oVMegQQNmR1sEttVb#:~:text=,) [[39]](file://file-D3gJ6oVMegQQNmR1sEttVb#:~:text=,or) orchestrator.py

<file://file-D3gJ6oVMegQQNmR1sEttVb>

[[8]](file://file-PAkVozqRNNKExGKLMyRB8r#:~:text=for%20oi%20in%20range%28len%28getattr%28task%2C%20,%3C%3D%2015) [[33]](file://file-PAkVozqRNNKExGKLMyRB8r#:~:text=def%20build_option_card_ids%28new_cards%3A%20List,first%20PRO%20card%20per%20option) [[34]](file://file-PAkVozqRNNKExGKLMyRB8r#:~:text=def%20dedup_per_option%28cards%3A%20List%5BDict%5Bstr%2C%20Any%5D%5D%2C%20,scoped%20cards%20while%20preserving%20order) [[35]](file://file-PAkVozqRNNKExGKLMyRB8r#:~:text=by_opt%3A%20Dict,float) factcards.py

<file://file-PAkVozqRNNKExGKLMyRB8r>

[[9]](file://file-P38QvTZmhYeiMHmqAinDVQ#:~:text=,import%20re%20as%20_re) [[19]](file://file-P38QvTZmhYeiMHmqAinDVQ#:~:text=%23%20has%20option,for%20c%20in%20cited_cards) [[22]](file://file-P38QvTZmhYeiMHmqAinDVQ#:~:text=,get%29%20if%20witness_scores%20else%20None) [[23]](file://file-P38QvTZmhYeiMHmqAinDVQ#:~:text=,letter%20in%20opt_map%20else%20None) [[24]](file://file-P38QvTZmhYeiMHmqAinDVQ#:~:text=,get%29%20if%20witness_scores%20else%20None) [[27]](file://file-P38QvTZmhYeiMHmqAinDVQ#:~:text=This%20module%20isolates%20small%2C%20testable,) [[55]](file://file-P38QvTZmhYeiMHmqAinDVQ#:~:text=evidence) evidence.py

<file://file-P38QvTZmhYeiMHmqAinDVQ>

[[12]](file://file-Bn5VgZmZvLvkH6dC2teGdi#:~:text=%40dataclass%20class%20Quote%3A%20option%3A%20str,str%5D%20%3D%20None) [[14]](file://file-Bn5VgZmZvLvkH6dC2teGdi#:~:text=def%20adapt_cards_to_quotes%28cards%3A%20List%5BDict%5Bstr%2C%20Any%5D%5D%20,append%28q%29%20return%20out) evidence\_schema.py

<file://file-Bn5VgZmZvLvkH6dC2teGdi>

[[13]](file://file-9Fv1FZXe6jptdY185YjNN3#:~:text=Goal%3A%20extract%20discriminative%2C%20option%E2%80%91linked%20%E2%80%9Ccards%E2%80%9D,then%20cite%20them%20when%20answering) [[20]](file://file-9Fv1FZXe6jptdY185YjNN3#:~:text=LEARN%20rules%20,card%20quoting%20from%20a%20snippet) [[21]](file://file-9Fv1FZXe6jptdY185YjNN3#:~:text=USE%20%28answer%29%20JSON%20schema%20,...) [[28]](file://file-9Fv1FZXe6jptdY185YjNN3#:~:text=text%20via%20%60where.scope%3D,card%20quoting%20from%20a%20snippet) [[29]](file://file-9Fv1FZXe6jptdY185YjNN3#:~:text=Record%20shape%20%28MCQ%29%20%60%60%60%20,432%2C) [[38]](file://file-9Fv1FZXe6jptdY185YjNN3#:~:text=ICL_Simulator_Handbook,then%20cite%20them%20when%20answering) [[46]](file://file-9Fv1FZXe6jptdY185YjNN3#:~:text=,for%20reproducible%20runs) [[50]](file://file-9Fv1FZXe6jptdY185YjNN3#:~:text=,432%2C) [[51]](file://file-9Fv1FZXe6jptdY185YjNN3#:~:text=,and%20writes%20a%20summary%20CSV) [[54]](file://file-9Fv1FZXe6jptdY185YjNN3#:~:text=12%7D%2C%20,%5D%7D) ICL\_Simulator\_Handbook.md

<file://file-9Fv1FZXe6jptdY185YjNN3>

[[15]](file://file-6LARtihGXyBfCZiDWt2cuf#:~:text=def%20pass_pre_gates%28quotes%3A%20List,min_sources%3A%20return%20False) [[16]](file://file-6LARtihGXyBfCZiDWt2cuf#:~:text=def%20pass_post_gates%28quotes%3A%20List,keys%28%29%29%2C%20chosen_letter) [[17]](file://file-6LARtihGXyBfCZiDWt2cuf#:~:text=def%20evidence_health%28quotes%3A%20List,0%20srcs%20%3D%200) [[18]](file://file-6LARtihGXyBfCZiDWt2cuf#:~:text=,int%28srcs%29%2C) [[26]](file://file-6LARtihGXyBfCZiDWt2cuf#:~:text=def%20coverage_for_option%28quotes%3A%20List,ot) [[30]](file://file-6LARtihGXyBfCZiDWt2cuf#:~:text=_TOK%20%3D%20re.compile%28r%22%5BA) evidence\_gates.py

<file://file-6LARtihGXyBfCZiDWt2cuf>

[[36]](file://file-RTVkGhp9td6zcwN9SkVRfD#:~:text=window%20around%20hits%20for%20retrieval%2Fcontext,len%22%2C%20dest%3D%22min_card_len%22%2C%20type%3Dint%2C%20default%3D40) [[37]](file://file-RTVkGhp9td6zcwN9SkVRfD#:~:text=help%3D%22near,k%22%2C%20dest%3D%22per_option_top_k%22%2C%20type%3Dint) [[40]](file://file-RTVkGhp9td6zcwN9SkVRfD#:~:text=p.add_argument%28%22,citations%22%2C%20default%3DNone%2C%20choices%3D%5BNone%2C%20%22off%22%2C%20%22lenient) [[42]](file://file-RTVkGhp9td6zcwN9SkVRfD#:~:text=def%20main%28argv%3A%20list%5Bstr%5D%20,task%20type%20to%20run) [[43]](file://file-RTVkGhp9td6zcwN9SkVRfD#:~:text=%23%20Controllers%20p.add_argument%28%22,currently%20%27self%27%20only) [[44]](file://file-RTVkGhp9td6zcwN9SkVRfD#:~:text=help%3D,idk%22%2C%20dest%3D%22idk_enabled%22%2C%20action%3D%22store_true) [[45]](file://file-RTVkGhp9td6zcwN9SkVRfD#:~:text=,on) [[49]](file://file-RTVkGhp9td6zcwN9SkVRfD#:~:text=citation_mode%20%3D%20None%20if%20args,strict) [[52]](file://file-RTVkGhp9td6zcwN9SkVRfD#:~:text=if%20args.student%20%3D%3D%20,if%20args.notes_file) cli.py

<file://file-RTVkGhp9td6zcwN9SkVRfD>