

Structural Retrieval-Augmented Toolbox for Equity Long–Short Factor Portfolio Construction

NyoraTech

2025

Abstract

This paper outlines a fully transparent and automated toolbox for equity long–short portfolio construction from natural language description. The approach combines domain-specific in-context data, retrieval-augmented generation (RAG), and a structural retrieval pipeline designed for Barra-style ranked factor workflows. We describe how examples and specifications are written, how requests are transformed into indexable retrieval queries, how structurally relevant context is selected, and how large language models (LLMs) are evaluated on portfolio-construction code generation tasks. Coupled with latest advancement in LLM coding skills, the methodology guides LLM to produce code that is verifiable and testable and can be used in an Agentic equity stack. It represents a shift towards a fully transparent on-demand dynamic workflow compared to pre-compiled black box APIs.

1 Introduction

Equity long–short portfolio construction often involves translating research intent into reliable code: select a universe, compute or load factor scores, rank securities, neutralize or constrain exposures, size long and short books, and evaluate performance. In practice, users express these tasks in natural language, while production implementations require precise data schemas, factor definitions, risk-model conventions, and portfolio constraints.

Large language models can accelerate this translation, but they are sensitive to prompt context and may produce code that is syntactically plausible yet financially incorrect. Retrieval-augmented generation offers a way to ground generation in task-specific examples, API documentation, portfolio templates, and benchmark cases. This paper focuses on a short, technical system design for RAG-based code generation in the specific setting of Barra-style ranked factor equity long–short portfolios.

The paper is organized around three questions: how to write the in-context data, how to transform user requests into retrievable representations, how to retrieve structurally relevant examples.

2 Background

2.1 Retrieval-Augmented Generation

Retrieval-augmented generation separates the problem into two stages: retrieving relevant context from an external knowledge base and conditioning the LLM on that context while generating an answer [1]. For code generation, the retrieved context may include function signatures, data schemas, complete examples, unit tests, known failure cases, and domain rules. The core design challenge is that semantic similarity alone is often insufficient: two requests may use different words while requiring the same code structure, or similar words while requiring different portfolio logic.

2.2 Equity Long–Short Portfolio Construction

An equity long–short factor forms a portfolio with positive weights in securities expected to outperform and negative weights in securities expected to underperform. A typical ranked-factor workflow includes:

1. define the investment universe and eligible dates;
2. compute and combine signals;
3. rank securities cross-sectionally within date, sector, region, or other groups;
4. select long and short buckets, such as top and bottom deciles;
5. apply risk, exposure, turnover, liquidity, and leverage constraints;
6. produce weights, trades, diagnostics, and backtest metrics.

2.3 Barra-Style Factors

In this paper, Barra-style factors refer to normalized ranked signals placed on a common scale, for example from -3 to $+3$. The normalization step is important because raw accounting, price, risk, and fundamental variables may have different units, ranges, and outlier behavior. By converting each input signal into a comparable rank score, the system can combine multiple factors, compare securities cross-sectionally, and build portfolios from a common factor scale.

The proposed methodology covers the first four steps of the ranked-factor workflow above. The code must first define the eligible universe and dates, then compute or load the raw signals, then normalize and rank them within the relevant cross-section, and finally select the long and short buckets from the ranked scores. Later risk controls, such as exposure limits or turnover constraints, may still be added, but the core Barra-style factor construction is the normalized ranking pipeline that turns heterogeneous inputs into comparable portfolio-selection scores.

3 Problem Definition

The system receives a natural-language request such as: “construct a market-neutral long–short portfolio using a value rank, sector-neutral buckets, top and bottom deciles.” The expected output is query code (DuckDB) to produce the portfolio data and implements the requested investment logic.

The problem has three sources of ambiguity:

- **Domain ambiguity:** terms such as rank, neutralize, rebalance, beta, and exposure may refer to different implementation details.
- **Code ambiguity:** the same portfolio objective can be expressed through different SQL queries, function compositions, or data layouts.
- **Context ambiguity:** retrieved examples can be useful, irrelevant, or misleading if they match surface language but not structure.

4 Writing In-Context Data

The proposed solution applies context-engineering principles to the in-context corpus. In this paper, these principles are implemented as a structured record format for portfolio-construction examples rather than as a free-form prompt library. This choice addresses three known problems in long-context code generation: relevant information can be ignored when it is buried in the middle of the prompt, few-shot examples can bias the model toward copying surface forms, and unstructured schemas or examples can create ambiguity about the intended code path [2, 3, 4].

The practical design rules are therefore as follows. The context should be dense, explicitly delimited, and ordered by execution priority, following recent context-engineering guidance [5, 6, 7]. Schema information should be separated from examples and represented as verified metadata, not as raw unannotated SQL [16, 17, 18]. The active user request and normalized task specification should appear close to the generation step. Few-shot examples should explain the mapping from request to code rather than only showing final code. The example bank should vary factor names, schemas, syntax, and portfolio styles, and should include clearly labeled failure modes or anti-patterns. Finally, each record should store metadata that allows retrieval quality and generated-code correctness to be evaluated [20, 21, 22].

Our implementation follows these rules through a specification-first context format. The full generation context is not only a set of examples. It contains verified database schema information, examples retrieved by RAG, and explicit code-generation instructions. Within that larger context, each retrieved example follows the factor flow expected in the generated code: define the universe, compute indicators, then rank and select the portfolio. The raw user request is translated into a detailed specification that names the query type, required functions, columns, rebalance rule, constraints, and selection rule. The generated answer is then organized into three code blocks: `universe`, `indicators`, and `portfolio_logic`.

4.1 Proposed Context Format

The corpus design adapts the general context-engineering rules to this financial code-generation task. Task specifications become normalized portfolio specifications; schema notes become required input and output fields with short business annotations; constraint recipes become tagged operations; examples are retrieved dynamically rather than fixed statically; complete code is placed after the specification and template to reduce copying; and failure modes are stored as explicit checks. This makes the context compact, structured, and auditable.

At inference time, the LLM receives a complete context bundle with three ordered parts. First, the schema block defines the available tables, columns, joins, date fields, factor fields, and portfolio outputs. Second, the RAG block contains the selected examples, each written in the structured example format described in Section 4.2. Third, the instruction block states the required output interface, coding constraints, and validation expectations. The general context structure is:

```
generation_context:
  database_schema:
    tables: verified table names and business meanings
    columns: column names, types, units, and allowed uses
    joins: valid keys for securities, dates, factors, and exposures
  retrieved_examples:
    full_examples:
      - end-to-end context_record following the Section 4.2 structure
    universe_examples:
      - section-specific record for universe filters and joins
    indicator_examples:
      - section-specific record for factor computation and normalization
    portfolio_logic_examples:
      - section-specific record for ranking, buckets, and weights
  code_generation_instructions:
    output_format: three named code blocks universe, indicators, portfolio_logic
    constraints: preserve request intent and use only verified schema fields
    validation_targets: syntax, schema, rank direction, rebalance logic
```

The material described below belongs inside each example retrieved by RAG, not around the entire prompt. The retrieval library stores both full examples and section-specific examples for `universe`, `indicators`, and `portfolio_logic`. This makes retrieval modular: the system can retrieve a complete end-to-end portfolio example when useful, or retrieve only the block needed for a specific part of the target query. Each example record is stored with three parts: retrieval metadata, prompt-facing content, and validation metadata. The

retrieval metadata is used by the semantic and structural indexes; the prompt-facing content is the ordered material supplied to the LLM as an example; and the validation metadata defines the checks used to score retrieval and generation quality. The per-example envelope is:

```
context_record:
  metadata:
    record_id: stable identifier
    task_type: long_only_rank | long_short_rank | exposure_neutralization
    example_scope: full | universe | indicators | portfolio_logic
    difficulty: S | M | L
    tags: factor families, rebalance rule, selection rule, constraints
  retrieval_fields:
    request_text: original natural-language request
    canonical_query: normalized retrievable query object
    factor_flow: universe -> indicators -> portfolio_rank
    operation_tags: index_filter, rolling_window, zscore, rank, select
    query_template: abstract SQL or DataFrame pattern with placeholders
  prompt_context:
    user_request: raw request shown to the model
    detailed_specification: normalized portfolio specification
    plan: reusable implementation steps
    query_structure: abstract query layout
    code_example: optional complete implementation
  validation:
    required_columns: inputs and outputs expected in generated code
    financial_invariants: rank direction, rebalance dates, bucket sizes
    failure_modes: known anti-patterns for this task
    tests: executable or judge-based acceptance checks
```

All examples, including benchmark examples, use the same `prompt_context` structure. This keeps retrieval, generation, and benchmarking aligned around one auditable context layout.

4.2 In-Context Example Structure

Each in-context example has five ordered layers. The order is deliberate: the model first sees the user intent, then the normalized specification, then the reusable plan, then the query template, and only at the end, when needed, a complete code example. This structure is shared by full examples and by section-specific examples for `universe`, `indicators`, and `portfolio_logic`. For a section-specific example, the specification, plan, query structure, and optional code are simply scoped to that block. This creates a self-explanatory example: the model is shown why a block of code is needed before it sees the final implementation.

```
<user request>
filter SP500; rank by average of 30-month operating margin
and 30-month median debt-to-asset; rebalance monthly.
```

```
<detailed specification>
Universe:
  - index filter = SP500 |filter|
  - date filter after 2018 |filter|
Indicators:
  - rolling_mean(operating_margin, 30M) |indicators|
  - rolling_median(debt_to_asset, 30M) |indicators|
  - zscore, winsorise, average signals |indicators|
Portfolio:
```

```

- long-only top 30 percent by score      |logic|
- monthly rebalance                      |rebalance|
Outputs:
- universe columns: date, ticker, raw factors, sector
- code blocks: universe, indicators, portfolio_logic

<plan>
universe_filter -> window_functions -> standardise -> combine
-> monthly_rank -> top_percent_selection

<query structure>
Universe SELECT -> indicator CTEs -> monthly ranking SELECT

<code example>
optional complete implementation, placed after specification and template

```

In production, the prompt contains two to four examples in this format. These may be a mixture of one full example and several block-level examples. The first examples can stop at the specification and query-template layers; the final example may include complete code. Retrieved examples should be diverse rather than perfect duplicates of the query: variation in factor names, column names, rebalance rules, and portfolio styles discourages spurious correlations such as copying a fixed `LIMIT` clause or a stale date filter. The result is a short context that preserves the reasoning path from request to code without filling the prompt with redundant implementations.

4.3 Modular Block Retrieval

The example bank is designed for block-level composition rather than only monolithic retrieval. A full portfolio query can require universe filtering, indicator construction, normalization, signal combination, ranking, and portfolio selection. Storing only complete examples would require a different large template for every combination of these choices. Instead, the corpus stores reusable examples for the three output blocks: **universe** examples for filters, joins, dates, and eligible securities; **indicators** examples for rolling functions, winsorization, z-scoring, and signal combination; and **portfolio_logic** examples for rebalance dates, rank direction, bucket selection, long-short flags, and weights.

This design uses the compositional behavior of in-context learning. Prior work on compositional generalization and skills-in-context prompting suggests that LLMs can solve more complex tasks when the prompt exposes simple primitives and shows how they combine, rather than only showing large task-specific demonstrations [24, 25, 26]. For portfolio code generation, the primitives are not generic reasoning steps but verified SQL or DataFrame blocks with financial roles. The RAG system can therefore retrieve a universe primitive from one example, an indicator primitive from another, and a portfolio-selection primitive from a third, then ask the model to compose them under the active schema and instructions. This keeps the context smaller, improves structural diversity, and reduces the risk that the model copies an irrelevant full query when only one part of that query matches the user request [27].

4.4 Fields Stored for Retrieval

Each record stores semantic fields and structural fields for dual indexing:

```

record_id: unique identifier
request_text: original user request
task_type: long_only_rank | long_short_rank | exposure_neutralization
factor_flow: universe -> indicators -> portfolio_rank
universe_ops: index_filter, date_filter, liquidity_filter
indicator_ops: window_function, zscore, winsorise, combine

```

```

portfolio_ops: monthly_rebalance, dense_rank, top_percent_selection
required_columns: date, ticker, raw factor columns, sector, risk exposures
output_blocks: universe, indicators, portfolio_logic
query_template: abstract SQL pattern with placeholders
code_example: optional complete implementation
failure_modes: wrong rank direction, stale filter, missing rebalance,
               schema leakage, alpha-risk confusion
tests: executable checks or expected financial invariants

```

The natural-language fields support semantic search, while the operation tags, factor-flow fields, and query-template fields support structural search. The same record can therefore be retrieved when a user asks for “top decile by value” or “highest ranked companies by a composite accounting signal,” provided the underlying code pattern is similar. The achievement is that the corpus supports both retrieval precision and code-generation generalization.

5 From User Request to Indexable Representation

A user request is transformed into an indexable representation before retrieval. The goal is to preserve financial intent while normalizing superficial variation in wording.

5.1 Request Parsing

The parser extracts:

- portfolio objective: long-short, long-only, market-neutral;
- ranking signal: window function, group by;
- grouping logic: global, sector-neutral, industry-neutral, region-neutral;
- construction method: bucket selection, equal weighting;

5.2 Canonical Query Object

The extracted information is represented as a canonical query object. This object has two roles. It preserves the financial intent for semantic retrieval, and it exposes the implementation structure needed for the second, non-semantic index. In particular, the parser maps the request into a detailed specification and then into a query structure: which functions are required, how signals are combined, how normalization is applied, and how securities are selected.

```

intent: generate_portfolio_code
portfolio_type: equity_long_short
ranking_method: cross_sectional_rank
neutralization: sector_and_market
detailed_specification:
  universe: index_filter, date_filter, liquidity_filter
  indicators: rolling_window, signal_combination, normalization
  portfolio_logic: rebalance, rank, select_buckets, constrain, weight
query_structure:
  function_types: window_function, cross_sectional_zscore, dense_rank
  combination_type: average_standardized_signals
  normalization_type: winsorize_then_zscore
  selection_type: top_bottom_bucket_selection
abstract_code_pattern: load_data -> compute_indicators -> normalize
                      -> combine -> rank -> select -> validate

```

The natural-language fields are embedded for vector search, while the structured fields are indexed separately as operation tags and query-pattern tokens. The structural representation is therefore not another semantic embedding of the same words. It is a code-aware signature of the requested program shape: universe filters, function classes, rolling windows, signal-combination rules, normalization steps, ranking method, rebalance rule, and portfolio-selection logic.

6 Structural RAG Retrieval Pipeline

The retrieval pipeline implements the same guidelines at inference time [8, 9]. First, it keeps critical instructions and the active specification close to the generation step, addressing positional attention decay. Second, it avoids monolithic context dumps by retrieving only two to four structured examples. Third, it separates semantic matching from structural matching, so the system retrieves examples that are similar in both financial intent and implementation shape.

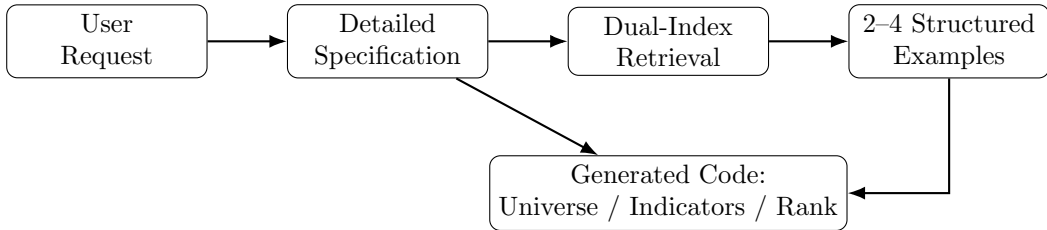


Figure 1: Specification-first workflow for retrieval-augmented portfolio code generation.

6.1 Stage 1: Request-to-Specification Translation

The first stage converts the user request into a normalized specification. For example, “filter SP500, rank by 30-month operating margin and debt-to-asset, and rebalance monthly” becomes a structured object containing universe filters, rolling-window functions, a composite signal, top-percent selection, monthly rebalance, required columns, and maximum lookback windows. The next step maps that specification into query-structure fields such as `window_function`, `rolling_median`, `standardize`, `combine_signals`, `monthly_rank`, and `top_percent_selection`. This implements the guideline of reducing semantic ambiguity before retrieval and makes the later structural match explicit.

6.2 Stage 2: Dual-Index Matching

The system uses two complementary indexes over the in-context records. The semantic index embeds the original request and explanatory text, allowing paraphrases to match. The structural index is different: it does not ask whether two requests use similar language, but whether they require the same query skeleton. It indexes operation tags and abstract query templates, such as `filter on index`, `window_function`, `rolling_aggregation`, `signal_combination`, `normalization`, `ranking on combined_function`, and `bucket_selection`. Retrieval combines both signals so that examples are selected because they describe a similar investment intent and because they require a similar code shape.

The structural score can be computed as overlap or weighted similarity over the extracted query-structure fields. Matches on portfolio type, rank direction, and selection rule receive high weight because they determine the output logic. Matches on function class, normalization type, and signal-combination type receive medium weight because they determine the indicator pipeline. Matches on incidental names, such as the literal factor column, receive lower weight unless the schema requires that exact field. This weighting lets the system retrieve an example about a different accounting signal when it uses the same rolling-window, normalization, combination, and selection pattern, while rejecting an example with similar wording but a different construction style.

This is consistent with the guideline of dynamic example retrieval rather than static few-shot prompting [19, 18]. The retrieval budget is intentionally small: for this task the system should prefer a few highly descriptive and structurally diverse examples over many near-duplicates. It also reduces shortcut learning: a request about “quality rank” and a request about “operating margin plus leverage” may use different language but share a template, while two requests mentioning the same factor may require different code if one is long-only and the other is long-short with exposure constraints.

6.3 Stage 3: Template-Guided Generation

The retrieved examples provide the model with a constrained output interface. The generator produces code in three sections. The `universe` section selects dates, tickers, and raw columns. The `indicators` section computes rolling functions, cross-sectional standardization, winsorization, and composite scores. The `portfolio_logic` section selects rebalance dates, ranks securities within each date, and assigns long or short flags. For iterative correction, the generated code can be sent to a separate validator or sandbox; parser errors, missing columns, or failed financial checks are fed back as a compact refinement context rather than appended as an unbounded log [10, 11, 12]. The final achievement is a context layout that is compact enough to avoid context bloat but explicit enough to preserve financial logic.

7 Generation and Verification

After retrieval, the LLM receives the user request, the canonical query object, selected context records, and explicit generation instructions. The generated code should be checked for:

- API validity and import correctness;
- required input and output schema compatibility;
- correct long-short sign convention;
- ranking direction and bucket selection;
- neutrality, exposure, and leverage constraints;
- reproducible tests or diagnostic assertions.

8 Overfitting Versus Generalization in In-Context Learning

Section 4 defines the proposed context format; this section explains why that format reduces overfitting and improves generalization. In-context overfitting occurs at inference time when the model treats retrieved examples as templates to copy rather than as evidence for an abstract rule. In portfolio code generation, this can produce executable but wrong SQL: the model may preserve a stale SP500 filter, reuse the wrong rebalance calendar, invert the rank direction, copy a long-only selection when a long-short book is requested.

The main failure modes follow the context-engineering literature [13, 14, 15]. Copying bias appears when complete code examples dominate the prompt, causing the model to reuse literal variables, constants, or query fragments. Spurious correlation appears when incidental details, such as a fixed date filter or `LIMIT` clause, are repeated across examples and then copied into unrelated outputs. Majority-pattern bias appears when most retrieved examples share the same construction style, such as top-decile long-only ranking. Formatting rigidity appears when the model preserves an example layout even when the new request requires a different logical structure. Recency bias appears when the last example controls the output even if the current request differs. Positional degradation appears when critical constraints are buried in a long prompt and are underused during generation [2, 3, 4].

The proposed solution implements the corresponding mitigations [4, 15]. The detailed specification and plan appear before code, making each example self-explanatory and reducing blind copying. Dynamic dual-index retrieval selects examples by both semantic intent and structural operations, reducing dependence on one static

demonstration set. The corpus stores task type, factor flow, operation tags, required columns, failure modes, and tests, so examples can be balanced across long-only, long-short, neutralized, and optimized variants. Negative or contrastive examples can also be retrieved when useful, for example an anti-pattern showing a missing rebalance filter or an incorrect rank direction. Finally, the prompt is kept short and ordered: active constraints and the current specification remain close to the generation instruction, while optional complete code appears only after the abstract mapping is established.

Generalization means that the model should reuse the workflow, not copy the literal example. For example, an example using operating margin and debt-to-asset can still be useful for a different accounting composite if the required structure is the same: filter the universe, compute rolling indicators, normalize scores, combine signals, rank monthly, and select the portfolio buckets. The retrieved context therefore stores both code and code structure, including operation tags and query templates, so the model can adapt the program shape to new factor names, columns, and selection rules.

The block design makes this easier than monolithic RAG. A monolithic retrieved query may contain useful indicator logic but irrelevant universe filters or portfolio rules, which encourages copying the wrong details. In the proposed format, the system can retrieve a `universe` block for eligibility rules, an `indicators` block for normalization and signal combination, and a `portfolio_logic` block for ranking and long-short selection. This lets the final query be composed from relevant pieces while the active specification controls how the pieces are joined. The final achievement is a controlled context architecture that makes generated portfolio code more faithful, testable, and reusable.

9 Datasets and Evaluation

We provide here the RAG examples described in Section 4. The datasets of potfolios description and generated code (S, M, and L versions) are released in a separate document. That document defines the dataset construction procedure, the examples included in each split, and the evaluation protocol for comparing LLMs with and without RAG.

10 Conclusion

This paper presents a compact system design for automated portfolio-construction Barra-style equity long-short workflows. By combining structured in-context data, canonical request parsing, dual-index retrieval, and AST-weighted reranking, the system aims to improve correctness and generalization for domain-specific code generation. The block structure of the retrieved examples mirrors the natural research workflow—universe definition, indicator construction, and portfolio logic—so the methodology can be tested in depth at both the component level and the full-query level. This modular structure also makes performance evaluation more informative, because failures can be localized to a specific retrieved block rather than treated as opaque errors in a monolithic generated query. The approach represents a shift from black box API towards a code generation that is open and accessible and understandable to all organisation levels.

References

- [1] Patrick Lewis et al. Retrieval-augmented generation for knowledge-intensive NLP tasks. *NeurIPS*, 2020.
- [2] Nelson F. Liu et al. Lost in the middle: How language models use long contexts. *TACL*, 2024.
- [3] Author. The good, the bad, and the debatable: A survey on the impacts of data for in-context learning. *ACL Anthology*, 2025.
- [4] Author. Debiasing in-context learning by instructing large language models how to follow demonstrations. *Findings of ACL*, 2024.
- [5] deepset. Context engineering: The next frontier beyond prompt engineering. Technical blog, 2025.
- [6] Anthropic. Effective context engineering for AI agents. Engineering blog, 2025.
- [7] LangChain. Context engineering for agents. Technical blog, 2025.
- [8] Amazon Web Services. Documentation best practices for RAG applications. AWS Prescriptive Guidance, 2025.
- [9] Kapa.ai. RAG best practices: Lessons from technical teams. Technical blog, 2025.
- [10] Author. Active context compression: Autonomous memory management in LLM agents. *arXiv preprint*, 2026.
- [11] Microsoft. Compaction in agent conversations. Microsoft Learn, 2025.
- [12] Arize AI. Context management in agent harnesses: memory, files, and subagents. Technical blog, 2025.
- [13] Author. Shortcut learning in in-context learning: A survey. *arXiv preprint arXiv:2411.02018*, 2024.
- [14] Author. How robust are LLMs to in-context majority label bias? *arXiv preprint arXiv:2312.16549*, 2023.
- [15] Author. Mitigating copy bias in in-context learning through neuron pruning. *arXiv preprint arXiv:2410.01288*, 2024.
- [16] Nirmalya Ghosh. Schema pruning for text-to-SQL: 93% less context, zero LLM calls. Technical blog, 2026.
- [17] Author. Rethinking schema linking: A context-aware bidirectional retrieval approach for text-to-SQL. *arXiv preprint*, 2025.
- [18] Mistral AI. Building a text-to-SQL conversion system with Mistral AI, Neon, and LangChain. Cookbook, 2025.
- [19] SQLAI. Enhancing AI accuracy for SQL generations using retrieval-augmented generation. Technical blog, 2025.

- [20] Promptfoo. Assertions and metrics for LLM output validation. Documentation, 2025.
- [21] DeepEval. LLM-as-a-judge: Evaluation techniques and best practices. Technical blog, 2026.
- [22] MLflow. LLM-as-a-judge evaluation for LLMs and agents. Documentation, 2025.
- [23] Evidently AI. LLM-as-a-judge: A guide to using LLMs for evaluations. Technical guide, 2025.
- [24] Le Wang et al. Diverse demonstrations improve in-context compositional generalization. *arXiv preprint arXiv:2212.06800*, 2022.
- [25] Denny Zhou et al. Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625*, 2022.
- [26] Author. Skills-in-context prompting: Unlocking compositionality in large language models. *Findings of EMNLP*, 2024.
- [27] Author. CodeChain: Towards modular code generation through chain of self-revisions with representative sub-modules. *arXiv preprint arXiv:2310.08992*, 2023.
- [28] MSCI. Barra equity risk model methodology reference to be added. Year.