A Strategic Curation Pipeline Using Minotaur

Think of this as a multi-stage filtering process, where each stage uses Minotaur's capabilities to increase the signal-to-noise ratio of your training data.

Stage 1: Foundational Quality Filtering (File-Level)

The first step is to establish a baseline of code quality and discard files that are syntactically incorrect or structurally uninformative.

1. **Syntax Validation:** The most fundamental filter. Use Minotaur to parse every single source file in the corpus against its identified language grammar. Any file that fails to parse due to a syntax error should be immediately discarded. This removes broken, incomplete, or malformed code that would only introduce noise into the training process.[1]
2. **Complexity Gating:** Not all valid code is useful for training. Use Minotaur to calculate code complexity metrics to filter out outliers.[2]
    - **Filter out low-complexity files:** Set a minimum threshold for metrics like cyclomatic complexity or Abstract Syntax Tree (AST) depth. This will discard trivial files like simple configuration scripts, single-function "hello world" examples, or boilerplate stubs that offer little learning value.
    - **Filter out high-complexity files:** Conversely, set an upper threshold to discard "spaghetti code." Files with excessively high complexity are often indicative of poor quality, anti-patterns, or are auto-generated, and are not representative of the clear, maintainable code you want your model to learn from.[2]
3. **Code Style and Readability Analysis:** Leverage static analysis to enforce a standard of code quality.[3] Minotaur can be programmed with rules to check for:
    - **Readability:** Flag and potentially discard code with poor readability, such as functions that are excessively long or have too many parameters.[1]
    - **Dead Code:** Use control-flow analysis to identify and filter out files containing significant portions of unreachable or dead code, which is a strong indicator of poor maintenance and quality.[3]

Stage 2: Addressing Language Evolution

This is a crucial step that standard data cleaning often misses. A corpus from GitHub contains a temporal smear of language versions (e.g., Python 2 vs. 3, C++98 vs. C++11 vs. C++20). Training on this mix can teach the model to generate outdated or syntactically mixed code.

1. **Version Identification via Grammar Features:** This is where Minotaur's grammar-aware parsing excels. Instead of simple keyword matching, you can identify language versions by the presence or absence of specific grammatical constructs.[4]
    - **Example (Python):** Configure Minotaur to detect Python 2 print statements vs. Python 3 print() functions, or the use of async/await which indicates Python 3.5+.
    - **Example (C++):** Detect the use of C++11 features like lambda expressions, auto type deduction, or range-based for loops.
2. **Corpus Stratification:** Once each file or project is tagged with a likely language version, you can create curated, version-specific sub-corpora. This allows for targeted training:
    - **Modern-Only Training:** To build a state-of-the-art transpiler, you can create a corpus consisting of only the most recent language versions (e.g., only Python 3.8+ and C++17+).
    - **Legacy Migration Training:** To build a tool for modernization, you can specifically isolate older code versions to train a model for that specific migration path.
3. **Library and Dependency Analysis:** Since Minotaur can load entire projects, it can parse dependency files (requirements.txt, pom.xml, package.json). This allows you to filter out projects that rely on obsolete or deprecated libraries, ensuring your model learns to use modern, current APIs.

Stage 3: Advanced Semantic Deduplication (Project-Level)

Standard deduplication only removes identical files. However, codebases are filled with *semantic clones*—functionally identical code that is implemented differently.[5] Training on this redundant information is inefficient.

1. **AST-Based Clone Detection:** Use Minotaur to parse code into ASTs. You can then implement or integrate algorithms that compare the *structure* of these trees, rather than just the text. This allows you to identify and remove Type-3 (structurally similar) and Type-4 (semantically similar) clones, creating a much more diverse and information-rich dataset.[5]
2. **Project-Level Filtering:** By analyzing the entire project, Minotaur can identify architectural smells, such as circular dependencies between modules. This allows you to filter out entire projects that are poorly structured, further improving the overall quality of the training data.

By implementing this pipeline, you transform the act of data preparation from simple cleaning into a strategic curation process. Minotaur would act as the core engine, allowing you to systematically address the challenges of quality and language evolution, ultimately producing a superior, refined corpus tailored to your specific training goals.