# GlobaLLM: Automated Open Source Contribution at Scale

Tom Rochette <tom.rochette@coreteks.org>

January 2, 2026 — af62e9d6

## 1 Introduction

Consider the following dilemma: you have unlimited access to state-of-the-art LLMs, but finite compute resources.

How do you maximize positive impact on the software ecosystem?

GlobaLLM is an experiment in autonomous open source contribution which attempts to address this question.

It's a system that discovers repositories, analyzes their health, prioritizes issues, and automatically generates pull requests - all while coordinating with other instances to avoid redundant work.

The core insight isn't just that LLMs can write code; it's that *strategic prioritization* combined with *distributed execution* can multiply that capability into something genuinely impactful.

This article explains how GlobaLLM works, diving into the architecture that lets it scale from fixing a single bug to coordinating across thousands of repositories.

## 2 The GlobaLLM Pipeline: A High-Level View

GlobaLLM follows a five-stage pipeline:

```
Discover -> Analyze -> Prioritize -> Fix -> Contribute
```

### 2.1 1. Discover

The system begins by finding repositories worth targeting.

Using GitHub's search API, it filters by domain, language, stars, and other criteria.

Current methodology uses **domain-based discovery** with predefined domains (`ai_ml`, `web_dev`, `data_science`, `cloud_devops`, `mobile`, `security`, `games`), each with custom search queries combining relevant keywords.

The system then applies **multi-stage filtering**:

1. **Language filtering**: Excludes non-programming languages (Markdown, HTML, CSS, Shell, etc.)
2. **Library filtering**: Uses heuristics to identify libraries vs applications (checks for package files like `pyproject.toml`, `package.json`, `Cargo.toml`; filters out "awesome" lists and doc repos; analyzes descriptions and topics)
3. **Quality filtering**: Language-specific queries include testing indicators (pytest, jest, testing)
4. **Health filtering**: Applies health scores to filter out unmaintained projects
5. **Dependent enrichment**: Uses libraries.io API to fetch package dependency counts for impact scoring

Results are cached locally (24hr TTL) to avoid redundant API calls and respect rate limits.

The goal isn't to find every repository - it's to find libraries where a contribution would matter.

## 2.2 2. Analyze

Once a repository is identified, GlobaLLM performs deep analysis to determine **whether contributing is worthwhile**.

This gate prevents wasting resources on abandoned projects, hostile communities, or repositories where contributions won't have impact.

It calculates a **HealthScore** based on multiple signals:

- **Commit velocity**: Is the project actively maintained?
- **Issue resolution rate**: Are bugs getting fixed?
- **CI status**: Does the project have passing tests?
- **Contributor diversity**: Is there a healthy community?

It also computes an **impact score** - how many users would benefit from a fix, based on stars, forks, and dependency analysis using `NetworkX`.

Repositories with low health scores or minimal impact are deprioritized or skipped entirely.

## 2.3 3. Prioritize

The system fetches open issues from approved repositories and ranks them using a sophisticated multi-factor algorithm.

Each issue is analyzed by an LLM to determine:

- **Category**: bug, feature, documentation, performance, security, etc.
- **Complexity**: 1-10 scale (how difficult to solve)
- **Solvability**: 0-1 score (likelihood of automated fix success)
- **Requirements**: affected files, breaking change risk, test needs

The prioritization then combines four dimensions:

**Health (weight: 1.0)**: Repository health adjusted for complexity.

A healthy repository with simple issues scores higher than an unhealthy repository with complex ones.

**Impact (weight: 2.0)**: Based on stars, dependents, and watchers.

Uses log-scale normalization (stars / 50,000, dependents / 5,000).

**Solvability (weight: 1.5)**: LLM-assessed likelihood of successful resolution.

Documentation and style issues (~0.9) rank higher than critical security (~0.3) due to automation difficulty.

**Urgency (weight: 0.5)**: Category multiplier × age × engagement.

Critical security bugs get 10× multiplier, documentation gets 1×.

The final formula:

```
priority = (health × 1.0) + (impact × 2.0) + (solvability × 1.5) + (urgency × 0.5)
```

Budget constraints filter the ranked list:

- Per-repository token limit (default: 100k)
- Per-language issue limit (default: 50)
- Weekly token budget (default: 5M)

Results are saved to the issue store with full breakdowns for transparency.

### 2.4 4. Fix

GlobaLLM claims the highest-priority unassigned issue and generates a solution.

This is where LLMs do the heavy lifting.

The `CodeGenerator` class sends a structured prompt to Claude or ChatGPT with:

- The issue title and description
- Repository context (code style, testing framework)
- Language-specific conventions
- Category-specific requirements (bug vs feature vs docs)

The LLM responds with a complete solution:

- **Explanation**: Step-by-step reasoning
- **File patches**: Original and new content for each modified file
- **Tests**: New or modified test files

The system tracks tokens used at every step for budget management.

### 2.5 5. Contribute

The final stage uses `PRAutomation` to create a well-structured pull request with context, tests, and documentation.

For trivial changes (typos, version bumps), it can even auto-merge.

## 3 Where LLMs Are Used

LLMs are the engine that powers GlobaLLM, but they're used strategically rather than indiscriminately.

**Stage 3 - Prioritize**: The `IssueAnalyzer` calls an LLM to categorize each issue.

Input: title, body, labels, comments, reactions.

Output: category, complexity (1-10), solvability (0-1), breaking_change, test_required.

This costs ~500 tokens per issue and feeds directly into the priority scoring.

**Stage 4 - Fix**: The `CodeGenerator` uses an LLM to generate complete solutions.

Input: issue details, repository context, language style guidelines.

Output: explanation, file patches (original + new content), test files.

This costs 1k-10k tokens depending on complexity.

The key insight: LLMs are only used for tasks requiring intelligence.

Discovery, health scoring, impact calculation, and PR automation use deterministic algorithms.

## 4 Scaling GlobaLLM

The real power of GlobaLLM emerges when you run multiple instances in parallel.

### 4.1 Distributed Agent Architecture

Each GlobaLLM instance has a unique `AgentIdentity`.

When it's ready to work, it calls:

```
globallm assign claim
```

This atomically reserves the highest-priority unassigned issue.

The assignment is stored in PostgreSQL with a heartbeat timestamp.

### 4.2 Issue Assignment System

To prevent multiple agents from working on the same issue:

1. Issues are marked `assigned` with an agent ID and timestamp
2. Heartbeats update every 5 minutes
3. If a heartbeat expires (30 minutes), the issue is reassigned

This allows crash recovery: if an agent crashes mid-work, another will pick up the issue.

### 4.3 Crash Recovery

The heartbeat system is elegant in its simplicity:

```
# Agent side
while working:
    update_heartbeat(issue_id, agent_id)
    do_work()

# Recovery side
expired = get_issues_with_expired_heartbeats()
for issue in expired:
    reassign(issue)
```

No distributed consensus needed - PostgreSQL's row-level locking handles contention.

### 4.4 Database Design

PostgreSQL is the central state store:

- **Connection pooling**: 2-10 connections per process (`psycopg` pool)
- **JSONB columns**: Flexible schema for repository/issue metadata
- **Indexes**: On frequently queried fields (stars, health_score, assigned status)
- **Migrations**: Versioned schema

## 5 Future Work

GlobaLLM is an evolving experiment.
There are numerous challenges that developers face on a daily basis that a system such as GlobaLLM will also encounter and would need to deal with in order to be more effective:

- Parallelizing work across multiple agents without conflicts or redundant effort.
- Producing "mental models" of repositories to better understand their goals, architecture, dependencies and trajectories.
- Have a higher-level decision-making system that can reason about which repositories to focus on based on broader trends in the open source ecosystem.
- Make decisions such as which programming languages to focus on.
- Have the ability to work with closed source repositories which may not have the same signals as open source ones (e.g., forks, stars, dependency count).

# 6 Conclusion

GlobaLLM is an experiment in what's possible when you combine LLM code generation with principled decision-making and distributed execution.

The goal isn't to replace human contributors - it's to handle the long tail of maintenance work that no one has time for, freeing up humans to focus on the interesting problems.

The system is actively developed and evolving.

Current work focuses on better prioritization heuristics, more sophisticated validation, and integration with additional LLM providers.

If you're interested in contributing or just want to run it yourself, the code is available on GitHub.

This system is far from perfect, but it's a step toward harnessing AI to make open source software healthier and more sustainable at scale.

It's also a way to explore what it looks like to have to make decisions at the scale of millions of repositories and billions of issues.