vibe-coder: A Single-File, Stdlib-Only Open-Source Coding Agent for Local LLMs

Technical Report — Architecture, Evolution, and Future Directions

Yoichi Ochiai
`vibe-local project`
`https://github.com/ochyai/vibe-local`

February 2026 — v0.9.1 (542 tests, 5,399 lines)

Abstract

We present **vibe-coder**, an open-source coding agent implemented as a single Python file (5,399 lines) that communicates directly with Ollama-hosted local LLMs. Unlike proprietary coding assistants that require cloud accounts, Node.js runtimes, or proxy layers, vibe-coder uses only the Python standard library and achieves feature parity with commercial tools through 15 integrated tools, an agentic loop with automatic context compaction, and a terminal UI with CJK support. Over 12 iterative review rounds using parallel AI agent audits, we applied 155+ fixes covering security hardening, reliability improvements, Unicode correctness, and user experience. The system is validated by 542 automated tests and has been tested by external users on Windows, macOS, and Linux. We report on the architecture, the iterative AI-assisted development methodology, findings from user testing (including a Windows user bug report), and future directions toward 1.0 release.

1 Introduction

The emergence of AI-powered coding assistants—Claude Code [1], GitHub Copilot Chat, Cursor, and others—has transformed software development workflows. However, these tools share common limitations:

1. **Cloud dependency**: Require active internet connections and vendor accounts.
2. **Proprietary binaries**: Closed-source CLIs that cannot be audited or modified.
3. **Runtime overhead**: Often depend on Node.js, Electron, or other heavy runtimes.
4. **Cost barriers**: Subscription models that exclude hobbyists and students.
5. **Language bias**: English-centric UX with poor CJK input and display support.

vibe-coder addresses all five limitations with a design philosophy we call *radical simplicity*:

> 世界中の誰でもダウンロードして使える、オフラインで動く最高のコーディングエージェント
>
> *"The best coding agent in the world that anyone can download and use, running fully offline."*

1.1 Design Constraints

- **Single file**: The entire agent is one Python file (`vibe-coder.py`), copyable via `scp` or USB stick.
- **Stdlib only**: Zero external Python dependencies—no `pip install` required.
- **Direct Ollama**: Communicates with Ollama's OpenAI-compatible API; no proxy process needed.

- **Offline-first**: All functionality works without internet (except WebFetch/WebSearch).
- **CJK-native**: Full support for Japanese, Chinese, and Korean input, display, and search.

2 Architecture

2.1 System Overview

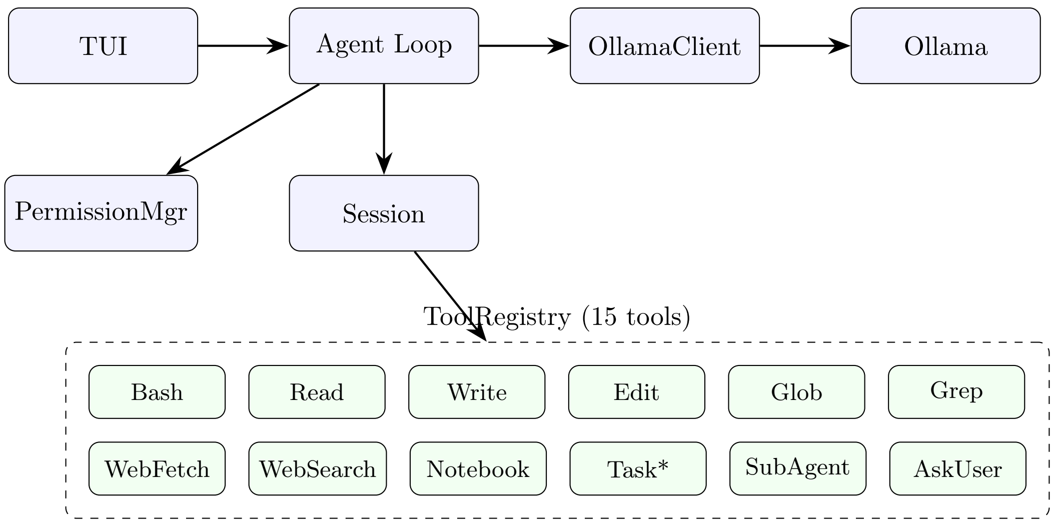

Figure 1: vibe-coder architecture: 9 core classes with 15 integrated tools.

2.2 Core Classes

Table 1 summarizes the 9 core classes.

Table 1: Core classes in vibe-coder.py

Class	Responsibility	~Lines
`Config`	CLI args, config file, env vars, model routing	150
`OllamaClient`	Ollama API (SSE streaming, function calling, retries)	250
`Tool` (base)	Abstract interface for all 15 tools	30
`ToolRegistry`	Tool dispatch, OpenAI schema generation	100
`PermissionMgr`	Permission rules (safe/ask/deny), pattern matching	200
`Session`	Message history, token estimation, compaction	400
`Agent`	Agentic loop: LLM → tool → result → loop	350
`TUI`	Terminal UI, readline, ANSI, streaming, CJK	500
`main()`	Entrypoint, Ollama connectivity, model selection	200

2.3 Tool Inventory

2.4 Agentic Loop

The core agent loop follows a standard ReAct [2] pattern:

```python
def run(self, user_input):
    self.session.add_user_message(user_input)
    while not self._interrupted:
        # 1. Stream LLM response
```

Table 2: 15 integrated tools with permission classification

Tool	Category	Permission	Description
BashTool	Execution	ask	Shell commands, timeout, background
ReadTool	File I/O	safe	Text, images (base64), PDF, notebook
WriteTool	File I/O	ask	Atomic write via `mkstemp`+`os.replace`
EditTool	File I/O	ask	Search-replace with uniqueness check
GlobTool	Search	safe	`os.walk` + `fnmatch`, mtime-sorted
GrepTool	Search	safe	Regex search, context lines, output modes
WebFetchTool	Network	ask	URL fetch, HTML→text, charset detect
WebSearchTool	Network	ask	DuckDuckGo HTML scraping, CJK locale
NotebookEditTool	File I/O	ask	Jupyter `.ipynb` cell editing
TaskCreateTool	Task mgmt	safe	Create tracked tasks
TaskListTool	Task mgmt	safe	List all tasks
TaskGetTool	Task mgmt	safe	Get task details
TaskUpdateTool	Task mgmt	safe	Update task status
SubAgentTool	Agent	ask	Spawn sub-agent (read-only, max 20 turns)
AskUserQuestionTool	UX	safe	Prompt user for clarification

```python
        response = self.client.chat(
            model=self.model,
            messages=self.session.get_messages(),
            tools=self.registry.get_schemas(),
            stream=True)
        text, tool_calls = self.tui.stream_response(response)

        # 2. XML fallback for models without function calling
        if not tool_calls and text:
            tool_calls = extract_tool_calls_from_text(text)

        self.session.add_assistant_message(text, tool_calls)
        if not tool_calls:
            break

        # 3. Execute tools (parallel for safe tools)
        results = self._execute_tools(tool_calls)
        self.session.add_tool_results(results)

        # 4. Context compaction if needed
        self.session.compact_if_needed()
```

Listing 1: Simplified agent loop

XML Fallback. Many local LLMs (especially Qwen-family models) sometimes emit tool calls as XML rather than structured function calling. We implement a robust XML extraction pipeline with: entity decoding, ReDoS bail-out for pathological patterns, JSON auto-parsing of arguments, known-tool filtering, and deduplication with JSON-normalized comparison.

Parallel Tool Execution. Safe tools (Read, Glob, Grep) are executed in parallel using `ThreadPoolExecutor`, with results buffered and displayed in call order to maintain deterministic output.

2.5 Context Management

Local LLMs typically have smaller context windows (4K–128K tokens) compared to cloud models. vibe-coder implements a multi-layered context management strategy:

1. **Token estimation**: CJK-aware character counting with expanded Unicode ranges (CJK Unified, Extension A, Hiragana, Katakana, fullwidth forms, Korean syllables).
2. **Explicit `num_ctx`**: The `options.num_ctx` parameter is sent in every Ollama API call to ensure server-side context window matches client-side tracking.
3. **Automatic compaction**: At 70% context utilization, older messages are summarized using a sidecar model while preserving the most recent 30 message pairs.
4. **Tool result truncation**: Large tool outputs are truncated at 50,000 characters with a notice.
5. **Max messages**: Hard cap at 500 messages with FIFO eviction preserving tool-call/result pairing.

2.6 Security Model

Security is designed in depth with multiple layers:

Table 3: Security measures by category

Category	Measures
Path traversal	Symlink resolution (`os.path.realpath`), protected path blocking (`/etc/passwd`, `/etc/shadow`, `.ssh/`), 10MB/50MB file size limits
SSRF	IP validation (private ranges, link-local, IPv4-mapped IPv6), DNS fail-closed, redirect host checking, localhost-only Ollama host
Injection	Environment variable sanitization (allowlist core + denylist secrets), dangerous command blocking (`rm -rf /`, `eval`, `base64 -d | sh`), prompt injection resistance
Atomic writes	All file operations use `tempfile.mkstemp` + `os.replace` for crash safety
Subprocess	`stdin=DEVNULL`, `start_new_session=True`, process group kill on timeout, zombie prevention

3 Iterative Development Methodology

3.1 AI-Assisted Multi-Agent Audit

A distinguishing feature of vibe-coder's development is the use of **parallel AI agent audits** for systematic code review. In each round, 3–40 specialized agents were launched concurrently, each focused on a specific concern (security, reliability, UX, testing, performance, etc.). Their findings were triaged, cross-referenced against the current codebase, and applied in batch.

3.2 Methodology Details

Each audit round follows a consistent process:

1. **Agent deployment**: Launch N parallel agents with specific audit charters (e.g., "find all SSRF vectors", "audit symlink handling", "test CJK display width").

Table 4: Development rounds and cumulative metrics

Round	Focus	Fixes	Tests	Lines
1	Foundation (20 agents)	29	–	∼1500
2	Robustness (16 agents)	15	–	∼2000
2.5	Edge cases	8	–	∼2200
3	IME + LLM + UX	16	–	∼2500
4	Deep audit (40 agents)	23	–	∼2800
5	Comprehensive (20 agents)	21	–	∼3200
6	Feature parity	21	119	∼3500
7	Security + reliability (4 agents)	29	160	∼3800
8	Features + critical fixes (5 agents)	22	384	∼4600
9	Robustness + UX	20+	432	∼4770
10	File/TUI/Ollama audit	20+	459	∼4916
11	Security deep audit	12	477	∼5000
12	Agent loop + install hardening	15+	542	5399

2. **Report collection**: Each agent produces a structured report with severity classifications (Critical, High, Medium, Low).
3. **Triage**: Cross-reference findings against current codebase; eliminate false positives and already-fixed issues.
4. **Implementation**: Apply fixes in dependency order (critical first, then high, medium, low).
5. **Validation**: Run full test suite after each batch; add regression tests for each fix.
6. **Documentation**: Update review-plan.md with applied fixes and updated metrics.

The multi-agent approach provides several advantages over sequential review:

- **Coverage**: Each agent explores a different concern axis, reducing blind spots.
- **Speed**: Parallel execution completes in the time of the slowest agent rather than the sum.
- **Diversity**: Different agent prompts surface different classes of issues.
- **Regression**: Accumulated tests prevent fix regressions across rounds.

4 CJK and Internationalization

4.1 Terminal Display Width

CJK characters occupy two terminal columns, while Latin characters occupy one. Standard `len()` is insufficient for terminal layout. We implement `_display_width()` using Unicode East Asian Width properties:

```python
def _display_width(text):
    w = 0
    for ch in text:
        eaw = unicodedata.east_asian_width(ch)
        w += 2 if eaw in ('W', 'F') else 1
    return w
```

Listing 2: CJK-aware display width calculation

4.2 Japanese IME Support

Japanese input via IME (Input Method Editor) requires double-Enter confirmation. The first Enter confirms the IME composition; the second submits the input. We detect CJK locales and adjust readline behavior accordingly.

4.3 Search Localization

DuckDuckGo searches are localized with the `kl` parameter (e.g., `kl=jp-ja` for Japanese) and `Accept-Language` headers, detected automatically from the system locale.

4.4 Permission Dialog Localization

Permission prompts accept Japanese responses: 常に (always), いいえ (no), 拒否 (deny), alongside English equivalents.

5 User Testing and Bug Reports

5.1 Case Study: Windows PowerShell (Issue #4)

A user (`rara2022`) reported two issues on Windows:

Bug 1: `$Debug` Parameter Collision. PowerShell reserves `$Debug` as a common parameter. Our launcher script (`vibe-local.ps1`) used `[switch]$Debug`, causing a conflict:

```
A parameter with the name 'Debug' was defined multiple times
```

Fix: Renamed to `$DebugMode` with corresponding internal variable updates.

Bug 2: `Invoke-RestMethod` Failure. The Ollama connectivity check used `Invoke-RestMethod`, which fails when the response is not valid JSON (Ollama's `/api/tags` can return unexpected formats). **Fix**: Replaced with `Invoke-WebRequest -UseBasicParsing` (checks HTTP status only) with a TCP socket fallback:

```
try {
    $resp = Invoke-WebRequest -Uri "$OllamaHost/api/tags" `
        -TimeoutSec 3 -UseBasicParsing -ErrorAction Stop
    return ($resp.StatusCode -eq 200)
} catch {
    # Fallback: TCP connection test
    $tcp = New-Object System.Net.Sockets.TcpClient
    $tcp.Connect($uri.Host, $uri.Port)
    $tcp.Close()
    return $true
}
```

Listing 3: Robust Ollama detection on Windows

5.2 Findings from User Testing

Key insights from user interaction and bug reports:

1. **Platform-specific pitfalls**: PowerShell reserved parameters and API behavior differences are not caught by Unix-only testing.
2. **Error message clarity**: Beginner users need actionable error messages with copy-pasteable commands (e.g., "`ollama pull qwen3:8b`").
3. **Installer robustness**: Package manager differences (Homebrew, apt, pacman, winget) require extensive platform detection.
4. **CJK display**: Banner art and separators must use Na-width characters to avoid misalignment on CJK terminals.
5. **Permission UX**: Users in non-English locales expect localized permission responses.

6 Performance Considerations

6.1 Startup Optimization

We applied 10 performance fixes to minimize startup latency:

- `GlobTool`: Primary `os.walk` instead of `Path.rglob` (avoids stat overhead).
- Schema caching: Tool schemas generated once at registry initialization.
- Deduplicated startup checks (single Ollama connectivity probe).
- Banner-first display (user sees output before model validation).
- `GrepTool`: Binary file probe (skip files >50MB or with null bytes).

6.2 Context Compaction Accuracy

Compaction preserves tool-call/result pairing integrity. Orphaned tool results (tool results without matching tool calls) are detected and removed during compaction to prevent LLM confusion.

7 Comparison with Existing Tools

Table 5: Feature comparison with commercial coding assistants

Feature	**vibe-coder**	**Claude Code**	**aider**	**Continue**
Offline operation		–	–	–
No cloud account		–	–	–
Single file		–	–	–
Stdlib only		–	–	–
Agentic loop			Partial	Partial
Tool use (15+)			Partial	Partial
CJK support		Partial	–	–
Context compaction				–
Session persistence				–
SubAgent			–	–
Image support			–	Partial
PDF reading			–	–
Task management			–	–
Plan mode			–	–
Auto model pull		N/A	–	–

8 Future Directions

8.1 Short-term (v1.0)

- **MCP Client**: Model Context Protocol support for external tool servers.
- **Persistent permissions**: Save per-project permission rules across sessions.
- **Rich diff preview**: Side-by-side colored diff display for Edit operations.
- **Type annotations**: Full type hints for IDE support and static analysis.
- **/doctor command**: Self-diagnostic tool checking Ollama, model availability, and system configuration.

8.2 Medium-term

- **Multi-model orchestration**: Route different tasks to different models (e.g., small model for Glob, large for code generation).
- **Hooks system**: User-defined pre/post hooks for tool execution.
- **Plugin architecture**: Loadable tool plugins without modifying the main file.
- **VS Code extension**: Integration as an IDE extension (language server protocol).
- **Benchmark suite**: Standardized coding task benchmarks for model comparison.

8.3 Research Directions

- **Compaction quality**: How does summarization quality affect long-session accuracy? Can smaller models produce adequate summaries?
- **Tool selection accuracy**: How do different local models compare in tool-use accuracy? What prompt engineering techniques improve reliability?
- **CJK token efficiency**: Local models have varying CJK tokenization; how does this affect context utilization?
- **Offline knowledge**: Without web search, how can local RAG (retrieval-augmented generation) supplement model knowledge?

9 Conclusion

vibe-coder demonstrates that a production-quality coding agent can be built as a single Python file with zero external dependencies. Through 12 rounds of iterative AI-assisted auditing, we achieved 155+ fixes and 542 tests while maintaining the single-file, stdlib-only constraint. The project validates a development methodology where parallel AI agents systematically audit different concern axes, with human oversight for triage and integration.

The system is freely available, fully offline-capable, and designed for global accessibility—including first-class support for CJK languages that are underserved by existing tools. We believe this approach—radical simplicity combined with rigorous iterative refinement—offers a viable path toward democratizing AI-powered development tools.

References

[1] Anthropic. *Claude Code CLI.* `https://docs.anthropic.com/`, 2024–2025.

[2] S. Yao, J. Zhao, D. Yu, et al. "ReAct: Synergizing Reasoning and Acting in Language Models." *ICLR*, 2023.

[3] Ollama. *Ollama: Run large language models locally.* `https://ollama.com/`, 2024.

[4] DuckDuckGo. *DuckDuckGo HTML Search.* `https://duckduckgo.com/`, 2024.