TamilTheni: A Case Study in AI-Assisted Development with Human Steering

Authors: Peoria Tamil School Development Team
Date: January 2026
Version: 1.0

Abstract

This paper documents the development of TamilTheni, a Tamil language learning web application built for the FETNA Tamil Theni competition. We employed an emerging development methodology we term "vibe coding with manual steering" — a collaborative approach where AI coding assistants generate code under continuous human guidance and oversight. This paper examines the project's intent, our development approach, the advantages and limitations encountered, and key learnings for teams considering similar methodologies.

---

1. Introduction

1.1 Project Background

The FETNA (Federation of Tamil Sangams of North America) Tamil Theni is an annual Tamil language competition that tests students across five categories: picture-to-word identification, sentence formation, translation, and word discovery using clues. Preparation for this competition traditionally relied on printed materials and in-person instruction.

TamilTheni was conceived to modernize this preparation by providing an interactive, accessible web application that students could use for self-study. The application needed to:

- Support all five competition categories
- Handle 800+ words with images and translations
- Work across devices (desktop, tablet, mobile)
- Be freely accessible without registration
- Support offline access via Progressive Web App (PWA) technology

1.2 Development Context

The development team consisted of volunteers with varying technical expertise. Time constraints necessitated rapid development without sacrificing quality. This context made AI-assisted development an attractive option.

---

2. Development Methodology: Vibe Coding with Manual Steering

2.1 Definition

"Vibe coding" refers to a development style where developers describe their intent in natural language, and AI assistants generate corresponding code. "Manual steering" adds an essential layer: continuous human oversight, review, and course correction of AI-generated outputs.

The methodology can be characterized as:

Aspect	Description
Intent Expression	Developers describe desired functionality in natural language
Code Generation	AI assistant produces implementation code
Review Cycle	Human reviews, tests, and validates outputs
Course Correction	Human redirects when AI produces incorrect or suboptimal solutions
Iterative Refinement	Continuous dialogue refines implementation

2.2 Workflow Structure

Our development workflow followed this pattern:

```
1. Human: Describe requirement or problem
      ↓
2. AI: Analyze codebase and propose solution
      ↓
3. Human: Review proposal, approve or redirect
      ↓
4. AI: Implement approved changes
      ↓
5. Human: Test and validate functionality
      ↓
6. Iterate: Refine based on testing results
```

2.3 Communication Style

Effective vibe coding required developing a communication style that balanced:

- **Specificity**: Enough detail for accurate implementation
- **Flexibility**: Room for AI to apply best practices
- **Context**: Background information for informed decisions
- **Constraints**: Clear boundaries and requirements

Example interaction:

> *Human: "Fix why words and images are not showing up in Theni 2."*
>
> *AI: [Analyzes code, identifies malformed HTML template causing querySelector failures, proposes fix]*
>
> *Human: [Validates fix works]*

---

3. Technical Implementation

3.1 Technology Stack

The project employed modern web technologies selected through collaborative decision-making:

Layer	Technology	Selection Rationale
Build Tool	Vite	Fast HMR, optimized production builds

Language	TypeScript	Static typing for maintainability
Styling	CSS (modular)	Simplicity, no framework overhead
Data	JSON	Structured, interoperable format
Deployment	GitHub Pages	Free, simple, reliable hosting

3.2 Architecture Evolution

The project architecture evolved through iterative refactoring sessions:

1. **Initial State**: Monolithic HTML files with embedded CSS and JavaScript
2. **First Refactor**: Extract CSS to separate files
3. **Second Refactor**: Convert JavaScript to TypeScript modules
4. **Third Refactor**: Create shared utility modules (timer, layout, audio)
5. **Final State**: Clean separation of concerns with documented architecture

This evolution was guided by human identification of maintenance pain points, with AI implementing the structural changes.

---

4. Advantages of the Approach

4.1 Rapid Prototyping

AI assistance dramatically accelerated initial development. Features that might take hours to implement manually could be scaffolded in minutes:

- Complete module implementations from requirements
- Complex data transformation scripts
- Comprehensive test suites
- Documentation generation

4.2 Reduced Cognitive Load

Developers could focus on "what" rather than "how":

- Pattern recognition delegated to AI
- Boilerplate generation automated
- Error handling patterns consistently applied
- Cross-file refactoring simplified

4.3 Knowledge Accessibility

The AI assistant served as an on-demand reference for:

- API documentation
- Best practice patterns
- Library usage examples
- Debugging strategies

4.4 Documentation Quality

AI-generated documentation was comprehensive and consistent:

- Architecture documents with visual diagrams

- Requirements documents with structured templates
- Code comments explaining intent
- README files with complete setup instructions

4.5 Refactoring Confidence

Large-scale refactoring became less risky:

- AI could analyze impact across the codebase
- Consistent application of changes
- Immediate identification of breaking changes
- Automated test generation for validation

---

5. Limitations and Challenges

5.1 Context Window Limitations

AI assistants have finite context windows, leading to:

- Incomplete awareness of large codebases
- Repeated explanations of project conventions
- Occasional inconsistencies across sessions
- Need for strategic context management

5.2 Verification Overhead

All AI-generated code required verification:

- Functional testing for correctness
- Visual inspection for UI changes
- Performance testing for efficiency
- Security review for vulnerabilities

5.3 Subtle Bugs

AI could introduce subtle issues not caught by cursory review:

- Malformed template strings (e.g., extra spaces in HTML)
- Incorrect edge case handling
- Off-by-one errors
- Browser compatibility issues

5.4 Tool Integration Challenges

Certain development workflows were problematic:

- Browser automation sometimes hung or failed
- Long-running processes could time out
- Environment-specific issues required manual intervention
- Complex debugging scenarios needed human insight

5.5 Learning Curve for Effective Prompting

Effective AI collaboration required learning:

- How to frame requests for best results
- When to provide more vs. less context

- How to redirect without starting over
- When to take manual control

---

6. Key Learnings

6.1 Human Steering is Essential

The "manual steering" component proved critical. Unsupervised AI coding would have produced:

- Inconsistent architectural decisions
- Unnecessary complexity
- Solutions that didn't fit the broader context
- Technical debt accumulation

Learning: AI is a powerful accelerator, not an autonomous developer.

6.2 Clear Checkpoints Improve Quality

Establishing verification checkpoints improved outcomes:

- Visual verification before moving on
- Test execution after changes
- Git commits as logical checkpoints
- Documentation review before finalization

Learning: Structured workflows prevent compounding errors.

6.3 Incremental Development Works Best

Large changes were risky; incremental development was safer:

- Small, focused requests produced better results
- Complex tasks benefited from decomposition
- Rollback was simpler with smaller changes
- Testing was more manageable

Learning: "Vibe" in small steps, not giant leaps.

6.4 Context Management is a Skill

Managing what the AI knows and remembers required strategy:

- Summarizing project state periodically
- Referencing relevant files explicitly
- Breaking conversations for fresh context
- Maintaining external documentation

Learning: Treat context as a resource to be managed.

6.5 Documentation Emerges Naturally

AI-assisted development naturally produced documentation:

- Explanations of changes were document-ready
- Architecture discussions crystallized into docs
- Requirements emerged from feature discussions

- Walkthroughs documented verification steps

Learning: Capture AI explanations as documentation.

6.6 Process Cleanup is Necessary

AI workflows could leave artifacts requiring cleanup:

- Hung processes from failed operations
- Temporary files from experiments
- Cache directories from Python tools
- Orphaned test files

Learning: Regular cleanup sessions maintain hygiene.

---

7. Recommendations for Teams

Based on our experience, we recommend:

1. **Start with Clear Architecture**
    - Establish conventions before heavy AI involvement
    - Document patterns for AI to follow
    - Create templates for common file types
2. **Implement Review Gates**
    - Never auto-apply AI changes without review
    - Test before committing
    - Verify complex changes manually
3. **Maintain Human Expertise**
    - Understand what the AI produces
    - Don't delegate decisions that require domain knowledge
    - Keep skills sharp for when AI fails
4. **Document Continuously**
    - Capture decisions as they're made
    - Keep requirements updated
    - Maintain architecture documentation
5. **Plan for Cleanup**
    - Schedule regular codebase hygiene sessions
    - Remove dead code promptly
    - Keep dependencies updated

---

8. Conclusion

The TamilTheni project demonstrates that AI-assisted development with human steering can be highly effective for building functional, maintainable applications. The approach dramatically accelerated development while maintaining quality, provided human oversight remained active.

The key insight is that "vibe coding" works best as a collaborative dialogue, not a delegation of responsibility. The human developer remains the architect, decision-maker, and quality arbiter, while the AI serves as an intelligent, tireless implementation assistant.

As AI coding tools continue to evolve, we anticipate this collaborative model becoming increasingly prevalent. Teams that develop effective human-AI collaboration patterns today will be well-positioned for the development methodologies of tomorrow.

Appendix A: Project Statistics

Metric	Value
Total Development Time	~2 months
Lines of TypeScript	~4,000
Lines of CSS	~1,500
Data Records	800+ words, 50 clues
Test Files	9 (unit + BAT)
Documentation Files	3 (README, ARCHITECTURE, REQUIREMENTS)

Appendix B: Technology Inventory

Category	Technologies
Frontend	TypeScript, HTML5, CSS3
Build	Vite, ESBuild
Testing	Vitest
Data Processing	Python, pandas
External APIs	Wikipedia, Gemini AI, Web Speech API
Deployment	GitHub Pages
