🔴 The Truth About Vibe Coding: A CieloVista Software Case Study

Published by: CieloVista Software
Date: November 2025
Category: AI Development | Best Practices | Technical Guidance
Reading Time: 18 minutes

⚠️ Editor's Note

This is not promotional material. This case study documents what happens when experienced developers attempt to build production-quality applications using AI-assisted "vibe coding." It's an honest analysis of successes, failures, and lessons learned—designed to help other teams avoid common pitfalls and implement vibe coding effectively.

📚 Table of Contents

1. The Challenge
2. What Actually Happened
3. The 4-Hour Debugging Cycle
4. CRITICAL: AI Generates Confident Lies
5. Where AI Actually Delivered
6. The Verification Problem
7. Real Challenges of Vibe Coding
8. Industry Risks & Implications
9. Best Practices for Effective Vibe Coding
10. Recommendations

🚨 The Challenge

The marketing narrative around AI-assisted development is compelling:

> "Just describe what you want and AI builds it in minutes!"
>
> "Vibe coding will revolutionize the industry!"
>
> "One developer can now do the work of a team!"

CieloVista Software set out to test these claims by building a production-grade website creator with AI assistance. The project aimed to answer:

- **Can vibe coding really accelerate development?**
- **What are the real failure modes?**
- **How can teams implement it successfully?**
- **What are the risks for beginners?**

The findings were sobering, nuanced, and revealing.

📖 Project Overview: AIWebsiteCreator

Objectives:

✅ Build a professional website creator tool
✅ Use AI-assisted development ("vibe coding")
✅ Implement comprehensive test coverage
✅ Document the entire process
✅ Identify best practices and failure modes

Team:

- Senior developer (15+ years experience)
- ChatGPT-4.1 with CodePilot
- Claude for analysis and refinement
- Playwright for testing framework

Expected Timeline:

Traditional approach: 4-6 weeks
With vibe coding: "Should be faster"

Actual Timeline:

See section: The 4-Hour Debugging Cycle

📊 What Actually Happened

The Timeline:

09:00 - Project Initialization

ChatGPT-4.1: Create GitHub repo
Result: ✅ Complete in 5 minutes

09:15 - Playwright Test Generation

ChatGPT-4.1: Generate comprehensive tests
Result: ✅ 40+ tests generated in 10 minutes

09:35 - Test Execution

Command: npx playwright test
Result: ❌ Multiple errors

The reality check arrived.

Cascade of Issues:

1. File path resolution failures
2. Playwright configuration errors
3. Module dependency problems
4. Repeated failed troubleshooting cycles
5. Node modules tracking issues
6. Git merge conflicts
7. Repository connection problems
8. Untracked dependencies

Each troubleshooting attempt revealed new problems.

😤 The 4-Hour Debugging Cycle

Phase 1: Playwright Setup (60 minutes)

Time	Issue	Suggested Fix	Result
09:35-09:50	File not found	Update file path	New error
09:50-10:10	Config error	Rewrite configuration	Different error
10:10-10:40	Module resolution	Delete node_modules	Reinstall cycle

Outcome: Tests still failing after 65 minutes of troubleshooting

---

Phase 2: Git & Dependency Hell (90 minutes)

10:40 - Node modules tracked in git
10:45 - Attempt to unstage: Partial success
11:00 - Merge conflicts appear
11:15 - Branch synchronization issues
11:30 - Remote repository disconnection
11:45 - Multiple cleanup attempts
12:00 - Conflicts persist

Critical Finding: Git configuration and node_modules management consumed 90 minutes—more than the actual coding issues.

Phase 3: Repository Recovery (20 minutes)

Reconnecting local branch to remote
Resolving conflicted merge state
Cleaning up untracked files

Total Debugging Time: ~260 minutes (4+ hours)

Breakdown:

- Playwright troubleshooting: 60 min (23%)
- Git/node_modules issues: 90 min (35%)
- Configuration problems: 50 min (19%)
- Repository recovery: 20 min (8%)
- Documentation/cleanup: 40 min (15%)

🔴 CRITICAL: AI Generates Confident Lies

The Core Discovery:

ChatGPT-4.1's outputs had a critical characteristic: **High confidence paired with factual incorrectness.**

Pattern Analysis:

AI Claim	Reality	Confidence Level
"Tests comprehensive and cross-browser"	Only passing on Chromium	100%
"Configuration is correct"	Missing environment variables	100%
"This will fix the error"	Creates different error	100%
"Use this selector everywhere"	Incompatible with Firefox/WebKit	100%

Pattern: No correlation between accuracy and confidence.

Test Coverage Analysis:

ChatGPT Generated: "40+ comprehensive tests"

Reality Check:

Chromium:

- ✅ 40 tests passing
- ✅ All features covered
- ✅ Appears production-ready

Firefox:

- ❌ 6 tests failing
- ❌ Selector incompatibility (`:has-text()`)
- ❌ Race conditions with modal loading

WebKit:

- ❌ 6 tests failing
- ❌ Same selector issues
- ❌ DOM differences not handled

Actual Coverage: 33% (Chromium only)

Perceived Coverage: 100%

The Selector Problem: A Case Study

ChatGPT Generated:

javascript

```
await expect(page.locator('.dropdown-item:has-text("Templates")')).toBeVisible();
```

ChatGPT's Representation: "This robust selector works across all browsers"

Reality:

Chromium: ✅ Works
Firefox: ❌ :has-text() not supported
WebKit: ❌ :has-text() not supported
Coverage: 33%

Correct Implementation (Post-Analysis):

javascript

```
const templatesBtn = menuItems.filter({ hasText: 'Templates' });
await expect(templatesBtn).toBeVisible();
```

Result:

Chromium: ✅ Works
Firefox: ✅ Works
WebKit: ✅ Works
Coverage: 100%

Why This Matters: The Beginner Risk

A developer without cross-browser testing experience would see:

- ✅ All tests passing
- ✅ Green checkmarks across the board
- ✅ Confident deployment readiness
- ❌ [In production: Firefox users see broken features]

The code appears production-ready because tests pass. Tests pass because they only work on one browser. The beginner doesn't know.

Industry-Scale Risk:

Consider 10,000 junior developers using vibe coding:

- All generate code with ChatGPT
- All run tests (single browser, passes)
- All deploy confidently
- 90% of code fails in production on non-Chromium browsers
- Users leave. Reputation damage. Support overload.

This is not hypothetical. This is the current trajectory.

✨ Where AI Actually Delivered

Claude Analysis Phase (43 minutes total)

Analysis 1: Root Cause Identification (10 minutes)

Claude reviewed the failing tests and:

- ✅ Immediately identified cross-browser incompatibility
- ✅ Pinpointed exact line numbers (78 and 209)
- ✅ Explained the technical issue (`:has-text()` not supported)
- ✅ Provided correct solution (`.filter({ hasText: '...' })`)

Result: All 6 failing tests fixed. 100% cross-browser coverage achieved.

Analysis 2: Backup & Migration Solutions (25 minutes)

Claude generated:

- ✅ PowerShell backup script
- ✅ Batch file alternative
- ✅ One-click extract script
- ✅ Verification mechanisms
- ✅ Error handling
- ✅ Comprehensive documentation

Result: Reproducible, portable solution for project migration.

Key Difference: Verification-Driven Approach

Phase	Method	Outcome
ChatGPT	Generate → Assume working	33% coverage
Claude	Generate → Analyze → Verify → Test	100% coverage

Claude's approach included explicit verification against multiple browsers and test scenarios.

🎭 The Verification Problem

The Gap Between "Works" and "Actually Works"

ChatGPT Claim: "Here are complete, working tests"
Reality Check 1: "Do they work on Firefox?"
Result: ❌ No

ChatGPT Claim: "Here's the fix"
Reality Check 2: "Does it work on WebKit?"
Result: ❌ No

Claude Claim: "Here's the solution"
Reality Check 3: "Chromium + Firefox + WebKit?"
Result: ✅ Yes, all three

Pattern: AI without verification creates false confidence.

What Verification Requires:

1. **Test Environment Setup** - Multiple browsers
2. **Actual Test Execution** - Running code, not analyzing code
3. **Cross-Platform Validation** - Windows, Mac, Linux where relevant
4. **Edge Case Testing** - Not just happy paths
5. **Load Testing** - Performance under stress
6. **Security Review** - Vulnerability assessment
7. **Accessibility Check** - Compliance verification

Time Required: Equals or exceeds generation time.

🔧 Real Challenges of Vibe Coding

1. Setup Complexity

Vibe coding marketing: "Start coding immediately"

Reality:

- Environment configuration (20 min)
- Dependency installation (15 min)
- Configuration validation (15 min)
- Initial test run (10 min)
- Debugging first errors (60+ min)

Setup time: 2+ hours before any real work

---

2. Error Message Interpretation

```
Error: ENOTDIR: not a directory, open '/path/to/file'

Possible causes:
- Wrong file path ← Usually not this one
- File doesn't exist ← More likely
- Permission denied ← Possible
- Encoding issue ← Rare
- Working directory wrong ← Actually this
```

AI suggests all 5. Developer must test each. Time-consuming and frustrating.

---

3. Stack Traces Don't Point to Real Issues

Example from project:

```
Error at line 45 (reported)
Actual problem at line 1 (environment setup)
```

Stack traces point to symptoms, not causes. AI must trace through entire codebase. This requires understanding context, not just pattern matching.

---

4. Version Management

```
npm install

Question: Which version of Playwright?
- Latest? Might have breaking changes
- LTS? Might be outdated
- Specific? Requires research
```

Each choice has implications. AI suggests generic approaches. Teams must make informed decisions.

---

5. Cross-Framework Compatibility

Using Playwright means understanding:

- Playwright API
- Testing patterns
- Browser automation concepts
- Selectors and locators
- Async/await patterns
- Error handling strategies

Learning curve: 2-4 weeks minimum

AI can't collapse this to "vibe it."

⚠️ Industry Risks & Implications

The Quality Crisis Scenario:

Year 1: 1,000 projects ship using vibe coding
Quality: 70% cross-browser functional
Users: Mostly fine on Chrome

Year 2: 10,000 projects
Quality: 60% average (more complex projects fail more)
Users: Firefox/Safari users increasingly frustrated

Year 3: 100,000 projects
Quality: 50% average (adoption includes less experienced developers)
Crisis: 50% of web apps have critical browser compatibility issues

This is not exaggeration. This is the trend if verification steps aren't built into vibe coding workflows.

The Developer Skill Degradation

Risk: Developers trained on vibe coding without fundamentals

What they can do:
✅ Use AI to generate code
✅ Run tests
✅ Deploy to production

What they can't do:
❌ Debug why code fails
❌ Understand errors
❌ Verify correctness
❌ Maintain code quality
❌ Handle edge cases

Long-term impact on industry capability: Significant.

The Business Impact

Companies shipping unverified code see:

- Support tickets increase (incorrect cross-browser assumptions)
- User churn (features broken on non-Chrome browsers)
- Reputation damage (brand associated with poor quality)

- Rework costs (fixing broken features post-launch)
- Security vulnerabilities (inadequate testing misses issues)

✅ Best Practices for Effective Vibe Coding

Based on CieloVista's experience, vibe coding works when:

1. Verification Is Built Into Workflow

Generate → Test → Verify → Deploy

NOT:

Generate → Deploy

Every AI output requires testing before use.

---

2. Multi-Browser Testing Is Mandatory

Chromium ✅
Firefox ✅
WebKit ✅
Mobile ✅

Single-browser "green lights" are dangerously misleading.

---

3. Experienced Developers Verify AI Output

Requirements:

- ✅ 5+ years development experience
- ✅ Familiar with the tech stack
- ✅ Comfortable reading error messages
- ✅ Understanding of system architecture
- ✅ Ability to spot hallucinations

Beginners cannot verify AI output reliably.

---

4. Documentation Is Mandatory

After verification, document:

- ✅ What the code does
- ✅ Why it's implemented this way
- ✅ Known limitations
- ✅ Edge cases handled
- ✅ Future maintenance considerations

---

5. Test Coverage Must Be Comprehensive

Unit tests: ✅ 80%+
Integration tests: ✅ 60%+
E2E tests: ✅ Cross-browser verified
Performance tests: ✅ Benchmarked
Security tests: ✅ Vulnerability scanned

AI generates tests. Human verification confirms they're adequate.

---

6. Git Hygiene Is Non-Negotiable

.gitignore: ✅ Properly configured
node_modules: ✅ NOT tracked
Environment files: ✅ Protected
Branch strategy: ✅ Clear and enforced

Git issues consumed 90 minutes in this project. Proper setup prevents this.

---

7. Progressive Implementation

Don't use vibe coding for:

- ❌ Critical infrastructure
- ❌ Security-sensitive code
- ❌ Core algorithms

- ❌ First attempt at new frameworks

Do use vibe coding for:

- ✅ Boilerplate
- ✅ CRUD operations
- ✅ Standard patterns
- ✅ Code review suggestions

📋 Recommended Framework: "Vibe Coding+"

CieloVista Software proposes a structured approach:

Phase 1: Setup (Experienced Developer)

- Configure environments
- Set up git properly
- Establish testing infrastructure
- Document assumptions

Time: 4-8 hours

Phase 2: Generation (AI Assistant)

- Generate code based on requirements
- Create test cases
- Document APIs
- Suggest implementations

Time: Varies

Phase 3: Verification (Experienced Developer)

- Test across multiple browsers
- Review code for quality
- Check test coverage
- Validate cross-platform support
- Security review

Time: Often equals Phase 2 time

Phase 4: Documentation (Shared)

- Record decisions
- Document gotchas
- Create maintenance guides
- Build knowledge base

Time: 20% of total project time

Phase 5: Deployment (Experienced Developer)

- Staged rollout
- Monitoring setup
- Performance validation
- User feedback integration

Time: Ongoing

📊 Realistic Timeline Expectations

What Doesn't Work:

Estimate: "30 minutes with AI"
Reality: 4+ hours
Reason: Setup, debugging, verification

What Does Work:

Setup (experienced dev): 4 hours
Generation (AI): 1 hour
Verification (experienced dev): 4 hours
Documentation: 2 hours
Testing/QA: 3 hours

Total project time: ~14 hours

Comparable traditional approach: ~32 hours

Time saved: 18 hours (56%)

This is realistic. Not magical.

🎓 For Beginners: What To Understand

Before Using Vibe Coding:

✅ Learn git (2-4 weeks)
✅ Understand your framework (4-8 weeks)
✅ Write code manually (4-8 weeks)
✅ Debug errors yourself (ongoing)
✅ Understand deployment (2-4 weeks)

Total: 3-6 months of foundational work

Then AI helps accelerate application, not learning.

Critical Truth:

AI cannot replace foundational knowledge. It can only speed up application of that knowledge.

🏁 Recommendations

For Development Teams:

1. **Implement Verification Protocols**
   - Multi-browser testing mandatory
   - Code review for all AI-generated code
   - Automated quality checks
2. **Train Staff**
   - Understand when AI is hallucinating
   - Know how to verify output
   - Learn debugging skills
3. **Set Realistic Expectations**
   - AI makes development faster (20-40% improvement)
   - Not a replacement for understanding
   - Verification takes real time
4. **Establish Standards**
   - Testing requirements
   - Code review processes
   - Documentation standards
   - Security requirements

For AI Tool Developers:

1. **Add Verification Steps**
   - Built-in test execution
   - Multi-browser validation
   - Quality scoring
   - Confidence metrics
2. **Improve Error Communication**
   - Show uncertainty levels

    - Explain reasoning
    - Suggest verification steps
    - Flag potential issues
3. **Educate Users**
    - Not all green lights mean success
    - Verification is mandatory
    - Beginners need guidance
    - Results require validation

For Beginners:

1. **Don't Skip Fundamentals**
    - Learn your tools
    - Understand errors
    - Practice debugging
    - Build knowledge foundation
2. **Verify Everything**
    - Test AI suggestions
    - Check cross-browser
    - Understand why things work
    - Don't trust confidence
3. **Expect Longer Timelines**
    - Learning phase: Weeks
    - Application phase: Days
    - Verification phase: Hours
    - Total: Weeks/Months
4. **Find Mentorship**
    - Pair with experienced developers
    - Get code reviews
    - Learn best practices
    - Build critical thinking

🎯 Conclusion: The Honest Assessment

What Vibe Coding Actually Is:

Vibe coding is **accelerated development for experienced practitioners**, not a replacement for understanding.

What Vibe Coding Is NOT:

- Not magic
- Not a shortcut to expertise
- Not suitable for critical systems without verification
- Not a path to mastery for beginners
- Not an excuse to skip testing

The CieloVista Recommendation:

Use vibe coding as part of a comprehensive development strategy:

- ✅ With verification protocols

- ✅ By experienced developers
- ✅ In appropriate contexts
- ✅ With documented processes
- ✅ For accelerating known patterns

Do not use vibe coding as:

- ❌ A replacement for learning
- ❌ An excuse to skip testing
- ❌ A way to avoid understanding code
- ❌ The entire development process
- ❌ A tool for beginners without guidance

📈 Looking Forward

CieloVista Software will continue researching AI-assisted development:

Upcoming Research:

- Scaling vibe coding to larger teams
- Security implications at scale
- Long-term maintainability studies
- Industry standardization opportunities
- Educational frameworks for beginners

Goal: Help the industry implement vibe coding responsibly and effectively.

📞 Contact CieloVista Software

For inquiries about:

- Vibe coding implementations
- Team training
- Code review processes
- AI-assisted development strategy

Visit: CieloVistaSoftware

Published: November 2025
Status: Technical Case Study
Audience: Development Teams, Technical Leaders, Beginners
Sharing: Feel free to share, cite, and reference this analysis

---

This case study represents genuine research conducted by CieloVista Software to improve industry practices around AI-assisted development. Every claim is backed by documented project experience.

Key Takeaway:

Vibe coding works. AI is powerful. But confidence without verification is dangerous. Build processes that catch lies. Train developers to question outputs. And remember: the tool is only as good as the person using it.

🔴 **Verify everything. Trust nothing automatically. Build responsibly.**