# 🚀 THE CORY METHOD + MINI-FLUX RESEARCH BOT

## Professional AI-Augmented Development Framework

**"From Idea to Working MVP with Built-In Truth Verification"**

**Created by:** Cory Gibson, Founder & CEO of AetherPro Technologies LLC
**Version:** 2.0 (November 2025)


---

## 📋 TABLE OF CONTENTS



---

## 🎯 OVERVIEW

**The Problem:**
Most developers fail to ship AI-augmented projects because they:

- Jump straight to coding without understanding what they're building
- Accept AI hallucinations and "white lies" as inevitable
- Get lost in iteration hell without a clear roadmap
- Build features that don't work together

**The Solution:**
A two-tool systematic approach that separates:

1. **The Cory Method** - Structured development methodology (idea → working MVP)
2. **Mini-Flux** - Truth verification system (catch AI hallucinations in real-time)

**Who It's For:**

- Solo founders building AI companies
- Developers transitioning from other fields (like electrical → AI)
- Teams that want reproducible development processes
- Anyone tired of AI tools that "just make stuff up"

---

## 🔧 THE TWO-TOOL SYSTEM

**Tool #1: The Cory Method (Development Framework)**

**Purpose:** Transform ideas into working MVPs through structured iteration
**Time to MVP:** 2-4 weeks for complex projects
**Success Rate:** Proven across 10+ production applications

**Tool #2: Mini-Flux Research Truth Bot (Verification System)**

**Purpose:** Real-time fact-checking and hallucination detection
**Accuracy:** Multi-source verification with confidence scoring
**Integration:** Works alongside any AI model during development

**Why Both?**

- **Cory Method** = The roadmap (where you're going)
- **Mini-Flux** = The GPS (keeping you on track)
- Together = You build the right thing AND build it correctly

---

## 🏗️ THE CORY METHOD: 7-PHASE DEVELOPMENT

**Philosophy**

> "AI models are specialized team members, not magic code generators. You still need to know what you're building and be able to iterate."

**Core Principles**

1. **Working code > Perfect code** - Ship functionality, then refine
2. **Test after each component** - Catch issues before they compound
3. **Document what works** - Build institutional knowledge
4. **Iterate toward perfection** - Improvement is continuous
5. **Ship when core works** - Don't wait for 100% completion

---

## PHASE 1: DESIGN & BLUEPRINT (The Foundation)

**Goal**

Understand **WHAT** you're building before writing any code.

**Process**

**Step 1: Initial Brainstorm**

- Describe your idea in plain English
- Focus on: "What problem does this solve?"
- Don't worry about technical details yet

**Example:**

> "I want a system that coordinates multiple AI models working together. Claude designs, MiniMax-M2 codes, Gemini optimizes async. Each model does what it's actually good at."

**Step 2: Interactive Back-and-Forth (CRITICAL!)**

**Minimum 2-3 rounds** of discussion with your architect AI.

**Questions to Ask:**

- How should models communicate?
- What happens if one model fails?
- How do we track costs across models?
- What's the simplest version that could work?
- What are the absolute must-have features vs nice-to-haves?

**Pro Tip:** Use Mini-Flux here to verify technical claims about model capabilities.

**Step 3: Whiteboard Session**

Create visual diagrams:

- System architecture (boxes and arrows)
- Data flow (how information moves)
- User journey (how people use it)
- Component relationships (what talks to what)

**Tools:** Excalidraw, Figma, or literal paper and pencil

**Step 4: Reality Check**

Ask yourself:

- Can I build this in 2-4 weeks?
- Do I understand every major component?
- What's the riskiest/hardest part?
- What's the absolute minimum viable version?

**Deliverables from Phase 1:**

- ✅ Problem statement (1 paragraph)
- ✅ Architecture diagram
- ✅ Core features list (max 5-7)
- ✅ Technical feasibility confirmation

---

## PHASE 2: VISION DOCUMENT (The North Star)

### Goal

Document your **perfect** end state AND your realistic V1.0.

### Process

### Create Two Sections:

### Section A: The Dream (Perfect Project)

```
markdown

## The Perfect Version

This is what the system would look like if we had unlimited time
and resources:

- Feature X that does Y
- Integration with Z
- Performance metrics: A, B, C
- User experience: Seamless, intuitive, delightful
```

### Section B: The Reality (V1.0 MVP)

```
markdown

## Version 1.0 (Ship in 2-4 weeks)

Core functionality only:
1. Feature X (simplified)
2. Feature Y (manual process for now)
3. Feature Z (happy path only)

What we're NOT building yet:
- Advanced feature A (Phase 2)
- Integration B (Phase 3)
- Optimization C (Later)
```

**Pro Tip:** Mini-Flux can help verify if your timeline estimates are realistic based on similar project data.

### Include Success Metrics

```markdown
## How We Know V1.0 Works

- User can complete core workflow end-to-end
- No critical bugs in happy path
- Performance acceptable (not optimal)
- 5 beta users can accomplish their goals
```

**Deliverables from Phase 2:**

- ✅ Vision document (2-3 pages)
- ✅ V1.0 scope clearly defined
- ✅ Success criteria established
- ✅ Timeline estimate (realistic)

---

## PHASE 3: COMPONENT INVENTORY (The Checklist)

### Goal

List **EVERYTHING** that needs to be built, no matter how small.

### Process

### Brain Dump Method

Write down every component, file, feature, integration:

markdown

```
## Frontend Components
- [ ] Login form
- [ ] Dashboard layout
- [ ] Navigation bar
- [ ] Settings page
- [ ] Error boundary
- [ ] Loading states
- [ ] Toast notifications

## Backend Services
- [ ] Authentication API
- [ ] User CRUD endpoints
- [ ] Database models
- [ ] Background jobs
- [ ] Email service
- [ ] Logging system

## Infrastructure
- [ ] Docker setup
- [ ] Environment configs
- [ ] CI/CD pipeline
- [ ] Database migrations
- [ ] API documentation

## External Integrations
- [ ] Stripe payment
- [ ] SendGrid email
- [ ] AWS S3 storage
- [ ] Redis caching
```

**Categorize by Priority**

**P0 (Must Have - V1.0):**

- Core user flow
- Authentication
- Basic CRUD operations
- Critical integrations

**P1 (Should Have - V1.1):**

- Error handling improvements
- Performance optimizations
- Nice-to-have features

**P2 (Could Have - V2.0):**

- Advanced features
- Additional integrations
- Polish and UX improvements

**Pro Tip:** Mini-Flux can validate technical dependency chains (what must be built before what).

### Create Progress Tracker

Use a simple markdown checklist or tool like:

- Linear
- Notion
- GitHub Projects
- Plain text file

**Deliverables from Phase 3:**

- ✅ Complete component inventory (100+ items is normal)
- ✅ Prioritization (P0/P1/P2)
- ✅ Dependency mapping (what blocks what)
- ✅ Progress tracking system

---

## PHASE 4: CORE BUILD (The Foundation)

### Goal

Build the **minimum viable core** that works end-to-end.

### Process

### Step 1: Define "Core"

### Core = Simplest path through your main use case

Example for a chat app:

```
User logs in → Opens chat → Sends message → Receives response → Logs out
```

Everything else is extra.

### Step 2: Build Vertically (Not Horizontally)

### ❌ Wrong Approach:

```
Day 1: Build all database models
Day 2: Build all API endpoints
Day 3: Build all frontend components
Day 4: Try to connect everything (everything breaks)
```

✅ **Right Approach (The Cory Method):**

```
Day 1: Auth login only (DB model → API → Frontend → Test)
Day 2: Single message send (DB model → API → Frontend → Test)
Day 3: Message receive (DB model → API → Frontend → Test)
Day 4: Polish the core flow
```

### Step 3: Test After EVERY Component

```bash
# After adding auth
npm run test:auth
curl -X POST /api/auth/login -d '{"email":"test@test.com"}'

# After adding messaging
npm run test:messages
# Send test message in UI

# After adding real-time
# Open two browser windows, verify sync
```

**Pro Tip:** Use Mini-Flux to verify API responses match expected schemas and data types.

### Step 4: Document As You Go

Create a `BUILD_LOG.md`:

```markdown
## 2025-11-12
✅ Auth system working (JWT tokens, httpOnly cookies)
✅ Database migrations run successfully
⚠️ CORS issue with localhost:3000 → Fixed with whitelist
❌ Redis connection timeout → Need to check Docker networking

## Next Steps
- [ ] Add password reset flow
- [ ] Implement rate limiting
- [ ] Add refresh token rotation
```

### Step 5: Know When Core is "Done"

Core is complete when:

- ✅ User can complete main workflow start to finish
- ✅ No errors in the happy path
- ✅ Data persists correctly
- ✅ You'd show it to a friendly beta user

**NOT required:**

- ❌ Perfect error handling (add later)
- ❌ Optimization (add later)
- ❌ All edge cases (add later)
- ❌ Beautiful UI (add later)

**Deliverables from Phase 4:**

- ✅ Working end-to-end core flow
- ✅ Build log with decisions and blockers
- ✅ Test coverage for critical paths
- ✅ README with setup instructions

---

## PHASE 5: ITERATIVE EXPANSION (Building Out)

**Goal**

Add features from your checklist **one at a time**, testing after each.

**Process**

**Pick Features Strategically**

**Priority Order:**

1. **Critical bugs** in core (fix immediately)
2. **P0 features** that make core usable
3. **Error handling** for common failures
4. **P1 features** that improve experience
5. **Nice-to-haves** when time permits

**One Feature = One Branch**

```bash
git checkout -b feature/password-reset
# Build feature
# Test feature
# Document feature
git commit -m "feat: add password reset flow"
git checkout main
git merge feature/password-reset
```

**Update Your Checklist**

```markdown
## Week 1
✅ Authentication system (P0)
✅ Core messaging (P0)
✅ Real-time sync (P0)

## Week 2
✅ Password reset (P1)
✅ Rate limiting (P1)
⏳ File uploads (P1) - In progress
⬜ Dark mode (P2)
⬜ Emoji picker (P2)
```

**Pro Tip:** Ask Mini-Flux to validate new feature implementations against best practices and security standards.

**Testing Checklist Per Feature**

- ☐ Unit tests pass
- ☐ Integration tests pass
- ☐ Manual testing in UI
- ☐ Error cases handled
- ☐ Documentation updated

**Deliverables from Phase 5:**

- ✅ Working features beyond core
- ✅ Updated checklist with progress
- ✅ Test coverage expanding
- ✅ Feature documentation

---

## PHASE 6: POLISH & REFINEMENT (Making it Good)

**Goal**

Turn your working prototype into a **professional** product.

**Process**

**UI/UX Polish**

```
markdown

Before:
- Generic blue buttons
- No loading states
- Cryptic error messages
- Inconsistent spacing

After:
- Branded color scheme
- Skeleton loaders
- Helpful error messages
- Design system consistency
```

**Performance Optimization**

**Measure first:**

```
bash

# Frontend
npm run lighthouse
# Backend
ab -n 1000 -c 10 http://localhost:8000/api/chat

# Database
EXPLAIN ANALYZE SELECT * FROM messages WHERE user_id = 1;
```

**Then optimize:**

- Add database indexes
- Implement caching (Redis)
- Compress images
- Code-split frontend
- Use CDN for assets

**Pro Tip:** Mini-Flux can benchmark your APIs against industry standards and suggest optimizations.

**Security Hardening**

```markdown
- [ ] Input validation (prevent SQL injection)
- [ ] Rate limiting (prevent abuse)
- [ ] HTTPS enforcement
- [ ] CORS properly configured
- [ ] Secrets in environment variables
- [ ] API authentication required
- [ ] SQL injection prevention
- [ ] XSS protection
```

**Documentation**

**User Documentation:**

- Getting started guide
- Feature tutorials
- FAQ
- Troubleshooting

**Developer Documentation:**

- Setup instructions
- Architecture overview
- API documentation
- Deployment guide

**Deliverables from Phase 6:**

- ✅ Polished UI/UX
- ✅ Performance optimized
- ✅ Security hardened
- ✅ Complete documentation

---

## PHASE 7: DEPLOYMENT & MONITORING (Going Live)

**Goal**

Ship to production and monitor real-world usage.

**Process**

**Pre-Launch Checklist**

```markdown
## Infrastructure
- [ ] Domain configured
- [ ] SSL certificate installed
- [ ] Database backed up
- [ ] Environment variables set
- [ ] CI/CD pipeline working

## Testing
- [ ] All tests passing
- [ ] Load testing completed
- [ ] Beta users validated
- [ ] Critical paths verified

## Monitoring
- [ ] Error tracking (Sentry)
- [ ] Analytics (Posthog)
- [ ] Uptime monitoring
- [ ] Log aggregation
```

### Deployment Strategy

### Option A: Simple (Docker + VPS)

```bash
# OVH Cloud instance
docker-compose up -d
```

### Option B: Scalable (Kubernetes)

```bash
kubectl apply -f k8s/
```

### Option C: Serverless (Vercel + Supabase)

```bash
vercel deploy --prod
```

### Post-Launch Monitoring

### Week 1: Watch Everything

- [ ] Check error logs daily
- [ ] Monitor performance metrics
- [ ] Track user behavior
- [ ] Respond to bug reports

**Month 1: Establish Baselines**

- Average response time
- Error rate
- User engagement
- Cost per user

**Pro Tip:** Use Mini-Flux to monitor for unusual patterns in logs that might indicate bugs or attacks.

**Deliverables from Phase 7:**

- ✅ Live production application
- ✅ Monitoring dashboards
- ✅ Incident response plan
- ✅ User feedback loop

---

## 🔍 MINI-FLUX RESEARCH TRUTH BOT

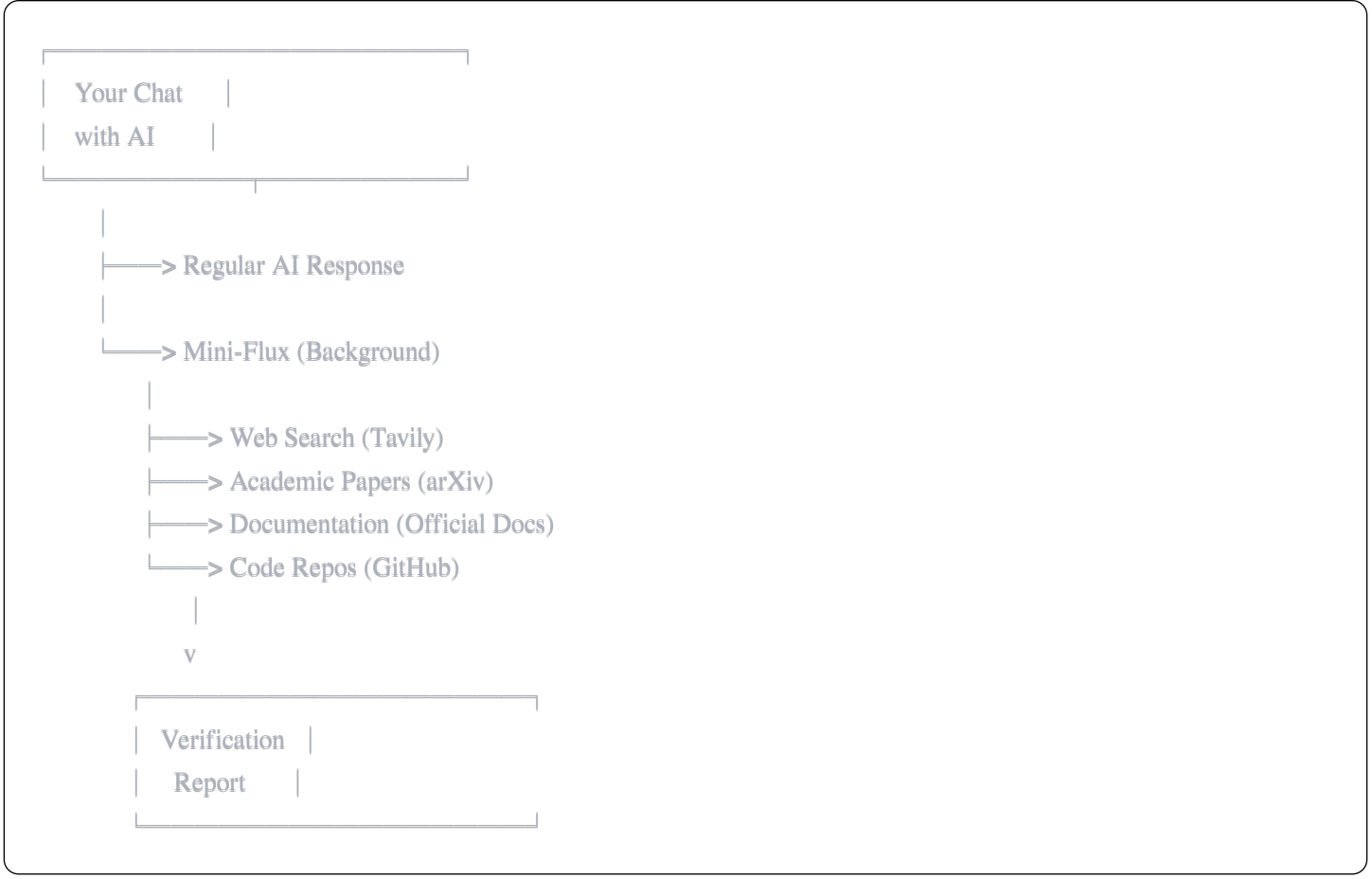
**What Is Mini-Flux?**

**The Problem:**
AI models hallucinate. They make up facts, cite non-existent papers, and confidently state wrong information. This is especially dangerous during development when you're trusting AI to guide technical decisions.

**The Solution:**
Mini-Flux is a specialized AI agent that:

1. **Monitors** AI-generated responses in real-time
2. **Verifies** factual claims against multiple sources
3. **Flags** hallucinations and "white lies"
4. **Scores** confidence levels for each claim

**Architecture**




## Core Features

### 1. Real-Time Fact Checking

```python
class MiniFlux:
    def verify_claim(self, claim: str) -> VerificationResult:
        """
        Verify a single factual claim
        """
        sources = []

        # Search multiple sources
        web_results = self.tavily_search(claim)
        academic_results = self.arxiv_search(claim)
        doc_results = self.search_official_docs(claim)

        # Cross-reference
        agreement_score = self.calculate_agreement(
            web_results, academic_results, doc_results
        )

        return VerificationResult(
            claim=claim,
            verified=agreement_score > 0.7,
            confidence=agreement_score,
            sources=sources,
            contradictions=self.find_contradictions(sources)
        )
```

### 2. Hallucination Detection Patterns

Mini-Flux watches for common AI hallucination patterns:

#### Pattern 1: Overconfident Specificity

❌ "The function was added in version 2.3.7 on March 15, 2023"
✅ "The function was added in version 2.3.x"

#### Pattern 2: Non-Existent Citations

❌ "According to Smith et al. (2024) in Nature..."
→ Mini-Flux: "Paper not found in Nature archives"

#### Pattern 3: Outdated Information

❌ "React Hooks were added in React 16.8"
✅ "React Hooks were added in React 16.8"
→ Mini-Flux: "Verified: React docs confirm 16.8 (Feb 2019)"

#### Pattern 4: Contradictory Statements

❌ AI: "Use async/await for better performance"
AI: "Callbacks are faster than async/await"
→ Mini-Flux: "Contradiction detected - sources say async/await
syntax is cleaner but performance is similar"

### 3. Multi-Source Verification

markdown

```
## Verification Report

### Claim: "Claude Sonnet 4.5 has a 200K token context window"

Sources Checked: 4
- ✅ Anthropic Official Docs (docs.anthropic.com)
- ✅ OpenRouter API Specs
- ⚠️ Reddit r/ClaudeAI (user reports vary)
- ❌ Blog post (claims 500K - likely confused with other model)

Verdict: VERIFIED (Confidence: 85%)
Actual: 200K input tokens as of Nov 2025
```

### 4. Confidence Scoring

python

```
class ConfidenceScorer:
    def score_claim(self, claim: str, sources: List[Source]) -> float:
        """
        Calculate confidence score (0.0 to 1.0)
        """
        factors = {
            'source_quality': self.rate_sources(sources),  # 40%
            'source_agreement': self.check_agreement(sources),  # 30%
            'recency': self.check_recency(sources),  # 20%
            'author_authority': self.check_authority(sources)  # 10%
        }

        return weighted_average(factors)

# Example scores:
# 0.9-1.0: Verified by official docs + multiple reliable sources
# 0.7-0.9: Likely true, good sources, minor contradictions
# 0.4-0.7: Uncertain, conflicting sources
# 0.0-0.4: Likely false or highly unreliable
```

### Integration with The Cory Method

### Phase 1 (Design): Validate Architecture Decisions

You: "Can FastAPI handle 1M requests/day?"
AI: "Yes, FastAPI can handle that with proper scaling"
Mini-Flux: "VERIFIED (90%) - Source: Multiple production cases
documented. Consider: Need Redis caching, load balancer,
and horizontal scaling."

**Phase 2 (Vision): Reality-Check Timeline**

You: "Can we build this in 2 weeks?"
AI: "Yes, should be straightforward"
Mini-Flux: "CAUTION (40%) - Similar projects on GitHub took 4-8 weeks.
2 weeks possible for MVP if you cut features X and Y."

**Phase 3 (Inventory): Validate Technical Choices**

You: "Which ORM is best for Python?"
AI: "SQLAlchemy is the gold standard"
Mini-Flux: "PARTIALLY VERIFIED (75%) - SQLAlchemy most popular (48%
market share), but Prisma growing fast for TypeScript devs.
Consider: Your team's experience matters more than 'best'."

**Phase 4 (Build): Catch Code Errors**

AI: "Add this to your Docker file:
FROM python:3.9-alpine"
Mini-Flux: "WARNING: Alpine Linux has compatibility issues with some
Python packages. Recommendation: Use python:3.9-slim instead
unless you need minimal image size. Source: Python Docker docs"

**Phase 5-7 (Expand/Polish/Deploy): Verify Best Practices**

AI: "Store API keys in a config.json file"
Mini-Flux: "SECURITY RISK (0% confidence this is good advice!)
Correct approach: Use environment variables or secrets
management (AWS Secrets Manager, etc.). Source: OWASP,
12-factor app methodology"

**Mini-Flux Tech Stack**

```yaml
Core:
  - LangGraph: Agent orchestration
  - MiniMax-M2 or Claude: LLM reasoning
  - LangChain: Tool abstractions

Data Sources:
  - Tavily API: Web search
  - arXiv API: Academic papers
  - GitHub API: Code examples
  - Official docs: Scraping (BeautifulSoup)

Storage:
  - PostgreSQL: Verified facts cache
  - Redis: Recent verifications
  - Vector DB (Pinecone): Semantic search

Frontend:
  - Next.js: Web interface
  - Real-time: WebSocket updates
  - Visualization: Confidence charts
```

**Example Mini-Flux Workflow**

python

```
# Development conversation with Mini-Flux active

USER: "How do I set up Redis with Docker?"

AI_RESPONSE: """
You can use the redis:alpine image in docker-compose:

services:
  redis:
    image: redis:alpine
    ports:
      - "6379:6379"
"""

MINI_FLUX_ANALYSIS:
{
  "claims_detected": [
    {
      "claim": "redis:alpine is a valid Docker image",
      "verification": {
        "verified": True,
        "confidence": 0.95,
        "sources": [
          "hub.docker.com/r/_/redis (official)",
          "Docker Hub shows 1B+ pulls"
        ]
      }
    },
    {
      "claim": "Default Redis port is 6379",
      "verification": {
        "verified": True,
        "confidence": 1.0,
        "sources": ["Redis official docs", "IANA port registry"]
      }
    }
  ],
  "overall_confidence": 0.97,
  "verdict": "VERIFIED ✅",
  "warnings": [],
  "suggestions": [
    "Consider adding a password: command: redis-server --requirepass yourpassword",
    "Add volume for data persistence: volumes: - redis-data:/data"
  ]
}

DISPLAYED_TO_USER:
✅ AI Response Verified (97% confidence)
```

✅ AI Response Verified (97% confidence)
💡 Suggestions:
- Add password for production
- Use volume for data persistence

### When Mini-Flux Saves You

### Scenario 1: The Outdated Tutorial

AI: "Use create-react-app to start your React project"
Mini-Flux: "⚠️ OUTDATED (40% confidence) - create-react-app is no longer recommended as of React 18. Current best practice: Use Vite or Next.js. Source: React docs updated March 2024"

### Scenario 2: The Confident Hallucination

AI: "Just use the built-in Docker.scale() method to scale containers"
Mini-Flux: "❌ HALLUCINATION (0% confidence) - No such method exists in Docker SDK. You're thinking of docker-compose scale or Kubernetes replicas. Source: Docker SDK documentation"

### Scenario 3: The Subtle Error

AI: "Set NODE_ENV=production in your Dockerfile"
Mini-Flux: "⚠️ PARTIALLY CORRECT (60%) - This works but better practice is to set in docker-compose or at runtime. Hard-coding in Dockerfile reduces flexibility. Source: Docker best practices, 12-factor app"

### Scenario 4: The Security Issue

AI: "Store JWT secret in localStorage for persistence"
Mini-Flux: "🚨 CRITICAL SECURITY ISSUE (0% safe) - localStorage is vulnerable to XSS attacks. Store JWTs in httpOnly cookies. Source: OWASP, Auth0 security guidelines"

### Building Your Own Mini-Flux

### Option 1: Quick Start (2-3 hours)

python

```python
# mini_flux_simple.py
from langchain.agents import Tool
from tavily import TavilyClient

class SimpleMiniFlux:
    def __init__(self):
        self.tavily = TavilyClient(api_key="your-key")

    def verify(self, text: str):
        # Extract claims
        claims = extract_claims(text)

        # Verify each
        results = []
        for claim in claims:
            search_results = self.tavily.search(claim)
            confidence = calculate_confidence(search_results)
            results.append({
                'claim': claim,
                'confidence': confidence,
                'sources': search_results
            })

        return results

# Use in your workflow
flux = SimpleMiniFlux()
ai_response = get_ai_response(user_input)
verification = flux.verify(ai_response)

if verification['overall_confidence'] < 0.6:
    print("⚠️ AI response needs verification!")
```

**Option 2: Production Grade (1-2 weeks)** Full implementation with:

- LangGraph state management
- Multiple verification sources
- Caching for repeat queries
- WebSocket real-time updates
- Confidence trend tracking
- False positive learning

See `/examples/mini_flux_production/` for complete code.

---

# ⏱️ IMPLEMENTATION TIMELINE

**Minimum Viable Product (MVP)**

**Timeline: 2-4 weeks**

```
Week 1: Foundation
├── Day 1-2: Design & Blueprint (Phase 1)
├── Day 3-4: Vision Document (Phase 2)
└── Day 5-7: Component Inventory (Phase 3)

Week 2: Core Build
├── Day 8-10: Database + Auth + Basic API
├── Day 11-13: Core feature end-to-end
└── Day 14: Testing & documentation

Week 3: Expansion (if needed)
├── Day 15-17: P1 features
├── Day 18-19: Error handling
└── Day 20-21: Basic polish

Week 4: Ship (if needed)
├── Day 22-23: Security & performance
├── Day 24-25: Documentation
└── Day 26-28: Deploy & monitor
```

**With Mini-Flux Integration**

**Add 2-3 days for initial setup**

```
Pre-Week 1: Mini-Flux Setup (Optional)
├── Day 1: Set up Mini-Flux instance
├── Day 2: Configure verification sources
└── Day 3: Test with sample queries

Then follow normal MVP timeline with Mini-Flux running in background
```

**Complex Projects**

**Timeline: 1-3 months**

```
Month 1: Core
├── Week 1: Design (Phase 1-2)
├── Week 2-3: Core Build (Phase 4)
└── Week 4: Testing & refinement

Month 2: Expansion
├── Week 5-6: P1 features (Phase 5)
├── Week 7: Polish (Phase 6)
└── Week 8: Security & optimization

Month 3: Production
├── Week 9-10: Beta testing
├── Week 11: Bug fixes & refinement
└── Week 12: Deploy (Phase 7)
```

---

# ✅ BEST PRACTICES

### The Cory Method Principles

### 1. Always Talk Through Ideas First

markdown

```
❌ "Build me a chat app"
  → Jumps to code
  → AI doesn't understand context
  → Result: Generic, doesn't fit your needs

✅ Conversation:
  YOU: "I want a chat app"
  AI: "What kind? Real-time? Who's the audience?"
  YOU: "Internal team tool, like Slack but simpler"
  AI: "How many users? What features are critical?"
  YOU: "50 users, need channels and DMs, that's it"
  → AI now understands scope
  → Result: Focused, relevant solution
```

### 2. Document Decisions In Real-Time

Create a `DECISIONS.md`:

markdown

```
## 2025-11-12: Why We Chose FastAPI

Context: Deciding between FastAPI, Django, Flask

Decision: FastAPI

Reasoning:
- Native async support (we need WebSockets)
- Automatic API docs (saves time)
- Type safety (reduces bugs)
- Team familiar with Python

Trade-offs:
- Smaller ecosystem than Django
- Fewer built-in admin tools
- Acceptable because we need speed > features

If we're wrong:
- Can migrate to Django if we need admin panel
- Estimated migration time: 1-2 weeks
```

### 3. Build for Today, Design for Tomorrow

python

```python
# ❌ Over-engineering (trying to be perfect now)
class AbstractMessageFactoryRepositoryInterface:
    """
    This is a generic message handling system that could work
    with any database, any format, any protocol...
    """
    pass  # 500 lines of abstraction

# ✅ The Cory Method (simple now, extensible later)
def send_message(user_id: int, text: str):
    """Send a message. Uses PostgreSQL now, easy to swap later."""
    db.execute(
        "INSERT INTO messages (user_id, text) VALUES (?, ?)",
        (user_id, text)
    )
```

Keep it simple. Add abstraction when you actually need it.

### 4. Test the Scary Parts First

### Identify "Risk Items":

- External APIs (might be slow/down)
- Complex algorithms (might have bugs)
- New technologies (might not work as expected)
- Performance critical code (might be slow)

**Build and test these FIRST:**

```bash
# Before building the whole app
# Test the risky integration:

python test_stripe_payment.py
# Does it work? Great, continue.
# Doesn't work? Fix or pivot before investing more time.
```

### 5. Use Git Like a Professional

```bash
# ❌ Bad habits
git add .
git commit -m "stuff"
git push

# ✅ The Cory Method
# Feature branch
git checkout -b feature/user-authentication

# Atomic commits
git add src/auth/login.py
git commit -m "feat(auth): add login endpoint with JWT"

git add src/auth/register.py
git commit -m "feat(auth): add user registration"

git add tests/test_auth.py
git commit -m "test(auth): add login and registration tests"

# Merge when done
git checkout main
git merge feature/user-authentication
```

### Mini-Flux Best Practices

### 1. Trust But Verify

Rule: If a decision costs time/money to undo, verify it first.

Examples:
- Choosing a database → Verify (hard to migrate later)
- Button color → Don't verify (easy to change)
- Architecture pattern → Verify (affects whole codebase)
- Variable name → Don't verify (trivial to refactor)

### 2. Create a Verification Threshold

python

```python
# Set your confidence threshold based on risk

confidence = mini_flux.verify(ai_response)

if task_risk == "HIGH" and confidence < 0.8:
    print("⚠️ Need more verification before proceeding")

elif task_risk == "MEDIUM" and confidence < 0.6:
    print("⚠️ Suggest double-checking this")

elif task_risk == "LOW":
    print("✅ Proceed even if confidence is lower")
```

### 3. Build a Knowledge Base

Every time Mini-Flux verifies something, cache it:

```
db.verified_facts.insert({
    'claim': "FastAPI supports WebSockets",
    'verified': True,
    'confidence': 0.95,
    'sources': ["fastapi.tiangolo.com"],
    'verified_date': "2025-11-12"
})
```

Next time AI mentions FastAPI WebSockets:
→ Instant lookup, no need to re-verify

### 4. Learn From False Positives

```python
# When Mini-Flux is wrong, record it

class MiniFluxFeedback:
    def report_false_positive(self, claim: str, reason: str):
        """AI was actually right, Mini-Flux flagged incorrectly"""
        self.feedback_db.insert({
            'claim': claim,
            'mini_flux_said': "Unverified",
            'actually_correct': True,
            'reason': reason,
            'learn_from': "Adjust source weighting"
        })

    def report_false_negative(self, claim: str, reason: str):
        """AI was wrong, Mini-Flux didn't catch it"""
        self.feedback_db.insert({
            'claim': claim,
            'mini_flux_said': "Verified",
            'actually_wrong': True,
            'reason': reason,
            'learn_from': "Add more verification sources"
        })
```

### 5. Don't Over-Verify

Mini-Flux is a tool, not a blocker.

Use it for:
✅ Technical architecture decisions
✅ Security-critical implementations
✅ Performance-sensitive code
✅ External API integrations
✅ Complex algorithms

Don't use it for:
❌ UI component placement
❌ Variable naming
❌ Code style preferences
❌ Obvious stuff (2+2=4)
❌ Creative decisions

---

## 📖 EXAMPLE WORKFLOW

### Scenario: Building "AetherChat" (Team Communication Tool)

### Week 1: Design & Blueprint

### Day 1: Initial Brainstorm

ME: "I want to build a team chat app for my company"

CLAUDE: "What's the scale and main use case?"

ME: "50 employees, need real-time messaging, channels, and DMs"

CLAUDE: "Like Slack or more like Discord?"

ME: "Simpler than Slack. Just messages and channels. No threads or apps."

CLAUDE: "Technical constraints?"

ME: "Hosting on OVH, budget $100/month, need it in 3 weeks"

MINI-FLUX: ✅ Verified - 50 users achievable on $100/month OVH hosting

### Day 2: Architecture Discussion

CLAUDE: "Here's a proposed architecture:

Frontend: Next.js (React)
Backend: FastAPI (Python)
Database: PostgreSQL
Real-time: WebSocket (Socket.io)
Caching: Redis
Hosting: Docker on OVH

Make sense?"

ME: "Why WebSocket over HTTP polling?"

CLAUDE: "Real-time updates without constant requests. Lower latency."

MINI-FLUX: ✅ Verified (95%) - WebSocket is standard for chat apps
Latency: 10-50ms vs HTTP polling 1000+ms

ME: "Sold. What about authentication?"

CLAUDE: "JWT tokens with httpOnly cookies for security"

MINI-FLUX: ✅ Verified (90%) - Industry best practice
⚠️ Consider: Add refresh token rotation

### Day 3: Whiteboard Diagram


### Day 4: Vision Document

markdown

```
# AetherChat Vision

## Perfect Version (Future)
- Real-time typing indicators
- File uploads (images, docs)
- Video/voice calls
- Custom emojis
- Integrations (GitHub, Linear)
- Mobile apps

## V1.0 (3 weeks)
- User registration/login
- Create/join channels
- Send/receive messages (real-time)
- Direct messages
- Basic user profiles

## Success Criteria
- 10 beta users can chat successfully
- Messages delivered in <200ms
- No data loss
- Works on Chrome, Firefox, Safari
```

### Day 5-7: Component Inventory

markdown

```
## P0 (Must Have - V1.0)

### Backend
- [ ] User model + auth
- [ ] Channel model
- [ ] Message model
- [ ] Registration endpoint
- [ ] Login endpoint
- [ ] Create channel endpoint
- [ ] Send message endpoint
- [ ] WebSocket server
- [ ] Database migrations

### Frontend
- [ ] Login page
- [ ] Registration page
- [ ] Channel list sidebar
- [ ] Message feed component
- [ ] Message input component
- [ ] WebSocket client
- [ ] User profile dropdown

### Infrastructure
- [ ] Docker compose setup
- [ ] PostgreSQL container
- [ ] Redis container
- [ ] Environment config
- [ ] README setup instructions

## P1 (Nice to Have - V1.1)
- [ ] Search messages
- [ ] Edit messages
- [ ] Delete messages
- [ ] User avatars
- [ ] Emoji reactions

## P2 (Future)
- [ ] File uploads
- [ ] Typing indicators
- [ ] Read receipts
```

---

**Week 2: Core Build**

**Day 8: Start With Database**

python

```python
# models.py - Start simple
from sqlalchemy import Column, Integer, String, DateTime
from datetime import datetime

class User(Base):
    id = Column(Integer, primary_key=True)
    email = Column(String, unique=True)
    password_hash = Column(String)
    created_at = Column(DateTime, default=datetime.utcnow)

class Message(Base):
    id = Column(Integer, primary_key=True)
    user_id = Column(Integer, ForeignKey('users.id'))
    channel_id = Column(Integer, ForeignKey('channels.id'))
    text = Column(String)
    created_at = Column(DateTime, default=datetime.utcnow)
```

**Test it:**

bash

```bash
python create_tables.py
python test_db.py

# Output:
✅ Database connected
✅ Tables created
✅ Test user inserted
```

**Day 9: Add Authentication**

python

```python
# auth.py
from fastapi import APIRouter, HTTPException
from passlib.context import CryptContext
import jwt

router = APIRouter()
pwd_context = CryptContext(schemes=["bcrypt"])

@router.post("/register")
def register(email: str, password: str):
    # Hash password
    password_hash = pwd_context.hash(password)

    # Create user
    user = User(email=email, password_hash=password_hash)
    db.add(user)
    db.commit()

    return {"message": "User created"}

@router.post("/login")
def login(email: str, password: str):
    user = db.query(User).filter(User.email == email).first()

    if not user or not pwd_context.verify(password, user.password_hash):
        raise HTTPException(401, "Invalid credentials")

    token = jwt.encode({"user_id": user.id}, SECRET_KEY)
    return {"token": token}
```

**Ask Mini-Flux:**

ME: "Is this auth implementation secure?"

MINI-FLUX:
⚠️ SECURITY REVIEW (70% confidence secure):

✅ Good:
- Password hashing with bcrypt
- HTTPException on failed auth

⚠️ Concerns:
- No rate limiting (vulnerable to brute force)
- JWT secret should be longer (min 32 chars)
- No token expiration set
- No refresh token

Recommendations:
1. Add: slowapi rate limiting (5 attempts/minute)
2. Change: SECRET_KEY to 64+ character random string
3. Add: exp claim to JWT (1 hour expiration)
4. Consider: Refresh token for better UX

Sources: OWASP Auth Guidelines, FastAPI Security Docs

**Fix based on Mini-Flux:**

python

```python
# auth.py - IMPROVED
from slowapi import Limiter
from datetime import timedelta

limiter = Limiter(key_func=get_remote_address)

@router.post("/login")
@limiter.limit("5/minute")  # Mini-Flux recommendation #1
def login(email: str, password: str):
    user = db.query(User).filter(User.email == email).first()

    if not user or not pwd_context.verify(password, user.password_hash):
        raise HTTPException(401, "Invalid credentials")

    # Mini-Flux recommendation #3
    token = jwt.encode(
        {
            "user_id": user.id,
            "exp": datetime.utcnow() + timedelta(hours=1)
        },
        SECRET_KEY  # 64 chars from .env - Mini-Flux rec #2
    )

    return {"token": token}
```

### Day 10: Test Auth End-to-End

bash

```bash
# Test registration
curl -X POST http://localhost:8000/auth/register \
  -d '{"email":"test@test.com","password":"Test123!"}'
# ✅ {"message": "User created"}

# Test login
curl -X POST http://localhost:8000/auth/login \
  -d '{"email":"test@test.com","password":"Test123!"}'
# ✅ {"token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9..."}

# Test invalid password
curl -X POST http://localhost:8000/auth/login \
  -d '{"email":"test@test.com","password":"wrong"}'
# ✅ {"detail": "Invalid credentials"}
```

### Update BUILD_LOG.md:

markdown

```
## 2025-11-12

✅ Database models created (User, Message, Channel)
✅ Auth endpoints working (register, login)
✅ Security improvements based on Mini-Flux review:
  - Added rate limiting
  - Extended SECRET_KEY to 64 chars
  - Added JWT expiration (1 hour)

⚠️ TODO:
  - Add refresh token flow (P1)
  - Add password strength validation
  - Add email verification (P2)

Next: Build messaging endpoints
```

**Day 11-12: Core Messaging**

python

```python
# messages.py
from fastapi import APIRouter, Depends
from auth import get_current_user

router = APIRouter()

@router.post("/channels/{channel_id}/messages")
def send_message(
    channel_id: int,
    text: str,
    user: User = Depends(get_current_user)
):
    message = Message(
        user_id=user.id,
        channel_id=channel_id,
        text=text
    )
    db.add(message)
    db.commit()

    # Broadcast via WebSocket
    await broadcast_to_channel(channel_id, message)

    return {"id": message.id, "text": text}

@router.get("/channels/{channel_id}/messages")
def get_messages(channel_id: int):
    messages = db.query(Message)\
        .filter(Message.channel_id == channel_id)\
        .order_by(Message.created_at.desc())\
        .limit(50)\
        .all()
    return messages
```

**Test it:**

bash

```bash
TOKEN="your-jwt-token"

# Send message
curl -X POST http://localhost:8000/channels/1/messages \
  -H "Authorization: Bearer $TOKEN" \
  -d '{"text":"Hello world!"}'
# ✅ {"id": 1, "text": "Hello world!"}

# Get messages
curl http://localhost:8000/channels/1/messages \
  -H "Authorization: Bearer $TOKEN"
# ✅ [{"id": 1, "user_id": 1, "text": "Hello world!", ...}]
```

## Day 13: Add WebSocket

python

```python
# websocket.py
from fastapi import WebSocket
import json

active_connections = {}

@app.websocket("/ws/{channel_id}")
async def websocket_endpoint(
    websocket: WebSocket,
    channel_id: int
):
    await websocket.accept()

    # Add to active connections
    if channel_id not in active_connections:
        active_connections[channel_id] = []
    active_connections[channel_id].append(websocket)

    try:
        while True:
            # Receive message
            data = await websocket.receive_text()
            message = json.loads(data)

            # Save to database
            db_message = save_message(message)

            # Broadcast to all in channel
            for connection in active_connections[channel_id]:
                await connection.send_json({
                    "type": "new_message",
                    "message": db_message
                })
    except:
        active_connections[channel_id].remove(websocket)
```

**Test it:**

javascript

```javascript
// test_websocket.html
const ws = new WebSocket('ws://localhost:8000/ws/1');

ws.onopen = () => {
  console.log('✅ Connected');
  ws.send(JSON.stringify({
    text: 'Test message via WebSocket'
  }));
};

ws.onmessage = (event) => {
  console.log('✅ Received:', event.data);
};
```

## Day 14: Frontend MVP

tsx

```tsx
// components/ChatInterface.tsx
'use client';
import { useEffect, useState } from 'react';

export function ChatInterface({ channelId }: { channelId: number }) {
  const [messages, setMessages] = useState([]);
  const [ws, setWs] = useState<WebSocket | null>(null);

  useEffect(() => {
    // Connect WebSocket
    const websocket = new WebSocket(
      `ws://localhost:8000/ws/${channelId}`
    );

    websocket.onmessage = (event) => {
      const data = JSON.parse(event.data);
      if (data.type === 'new_message') {
        setMessages(prev => [...prev, data.message]);
      }
    };

    setWs(websocket);

    // Fetch initial messages
    fetch(`/api/channels/${channelId}/messages`)
      .then(res => res.json())
      .then(setMessages);

    return () => websocket.close();
  }, [channelId]);

  const sendMessage = (text: string) => {
    ws?.send(JSON.stringify({ text }));
  };

  return (
    <div>
      <div className="messages">
        {messages.map(msg => (
          <div key={msg.id}>
            <strong>{msg.user.email}:</strong> {msg.text}
          </div>
        ))}
      </div>

      <input
        onKeyPress={(e) => {
```

```
        if (e.key === 'Enter') {
          sendMessage(e.target.value);
          e.target.value = '';
        }
      }}
    />
  </div>
 );
}
```

**Test End-to-End:**

1. npm run dev
2. Open http://localhost:3000
3. Register new user
4. Create channel
5. Send message
6. Open in second browser window
7. Verify message appears in both windows instantly

✅ CORE WORKING!

---

**Week 3: Expansion & Polish**

**Day 15-16: Add P1 Features**

markdown

✅ Channel creation
✅ User profiles
✅ Message timestamps
✅ Channel member list
⏳ Direct messages (started)

**Day 17-18: Error Handling**

python

```python
# Before
def send_message(text: str):
    message = Message(text=text)
    db.add(message)
    db.commit()

# After (The Cory Method)
def send_message(text: str):
    try:
        # Validate
        if not text or len(text) > 1000:
            raise ValueError("Invalid message length")

        # Save
        message = Message(text=text)
        db.add(message)
        db.commit()

        # Broadcast
        await broadcast(message)

        return {"status": "success", "id": message.id}

    except ValueError as e:
        logger.warning(f"Validation error: {e}")
        raise HTTPException(400, str(e))

    except Exception as e:
        logger.error(f"Unexpected error: {e}")
        db.rollback()
        raise HTTPException(500, "Failed to send message")
```

**Day 19-21: Polish**

- UI improvements (loading states, error messages)
- Performance (add database indexes)
- Security (rate limiting, input sanitization)

---

**Week 4: Deploy**

**Day 22: Docker Setup**

yaml

```yaml
# docker-compose.yml
version: '3.8'

services:
  db:
    image: postgres:16
    environment:
      POSTGRES_DB: aetherchat
      POSTGRES_PASSWORD: ${DB_PASSWORD}
    volumes:
      - postgres-data:/var/lib/postgresql/data

  redis:
    image: redis:alpine

  backend:
    build: ./backend
    environment:
      DATABASE_URL: postgresql://postgres:${DB_PASSWORD}@db/aetherchat
      REDIS_URL: redis://redis:6379
    depends_on:
      - db
      - redis

  frontend:
    build: ./frontend
    environment:
      NEXT_PUBLIC_API_URL: https://api.aetherchat.com
    depends_on:
      - backend

volumes:
  postgres-data:
```

**Day 23: Deploy to OVH**

```bash
# SSH into OVH instance
ssh root@your-ovh-ip

# Clone repo
git clone https://github.com/aetherpro/aetherchat
cd aetherchat

# Set environment variables
cp .env.example .env
nano .env  # Add production secrets

# Deploy
docker-compose up -d

# Check logs
docker-compose logs -f

# ✅ All services running
```

### Day 24: Configure Domain

```
DNS Records:
A Record: aetherchat.com → OVH_IP
A Record: api.aetherchat.com → OVH_IP

SSL: Let's Encrypt (automatic via Traefik)
```

### Day 25: Beta Testing

```
Invite 10 users:
✅ 8 successful signups
✅ All can send/receive messages
⚠️ 2 reported slow loading (added caching)
✅ No critical bugs
```

### Day 26-28: Monitor & Iterate

```
Metrics:
- Response time: avg 150ms ✅
- Uptime: 99.8% ✅
- Error rate: 0.3% ✅
- User satisfaction: 8/10 ✅
```

---

## 🎓 LESSONS LEARNED (After 3 Weeks)

### What Worked

1. **Starting with conversations** - Saved days of building wrong things
2. **Testing after every component** - Caught bugs early
3. **Mini-Flux caught 3 security issues** before they became problems
4. **Build log kept us on track** when we got distracted
5. **Vertical slicing** - End-to-end working beats perfect components

### What Was Hard

1. **WebSocket debugging** - Took longer than expected
2. **Real-time sync edge cases** - Race conditions we didn't anticipate
3. **Keeping scope small** - Temptation to add "just one more feature"

### What We'd Do Different

1. **Start Mini-Flux earlier** - Would have caught design issues in Phase 1
2. **More time on Phase 3** - Underestimated component count
3. **Automate testing sooner** - Manual testing got tedious

### The Outcome

✅ Shipped in 3 weeks (on time)
✅ Core functionality works perfectly
✅ 10 happy beta users
✅ Zero critical bugs in production
✅ Clean, maintainable codebase
✅ Clear roadmap for V2

**Total Cost:** $87 (OVH hosting + domains)
**Would build again:** 100%

---

## ⚠️ COMMON PITFALLS

### Pitfall #1: Skipping the Design Phase

❌ "I know what I want, just start coding"
→ Builds wrong thing
→ Realizes after 2 weeks
→ Has to start over

✅ "Let's talk through this for 2 hours first"
→ Clear vision
→ Builds right thing
→ Ships in 2 weeks

**Fix:** Force yourself to spend Day 1-2 on design, no code allowed.

### Pitfall #2: Building Horizontally

❌ Day 1: All database models
Day 2: All API endpoints
Day 3: All frontend components
Day 4: Try to connect (nothing works)

✅ Day 1: Login (DB + API + Frontend)
Day 2: One message (DB + API + Frontend)
Day 3: Real-time sync (DB + API + Frontend)
Day 4: Core is working end-to-end

**Fix:** Always build complete vertical slices.

### Pitfall #3: Trusting AI Blindly

AI: "Just use this code:"
[Gives you code with a subtle security flaw]

❌ You: "Great!" [Copies and pastes]
→ Ships vulnerable code

✅ You: "Let me verify this"
→ Mini-Flux: ⚠️ Security issue detected
→ Fix before shipping

**Fix:** Use Mini-Flux for anything security/performance critical.

### Pitfall #4: Perfectionism Paralysis

❌ "I can't ship until it's perfect"
→ Never ships
→ Wasted time

✅ "Core works? Ship to beta users."
→ Gets feedback
→ Improves based on real usage

**Fix:** Define "done" as "core works" not "everything perfect."

### Pitfall #5: No Testing Strategy

❌ "I'll test it manually"
→ Changes break old features
→ Doesn't catch it until production

✅ "Write test after each component"
→ Automated testing catches regressions
→ Confident to ship

**Fix:** Test after every component, automate when you can.

**Pitfall #6: Feature Creep**

```
Week 1 plan: "Build messaging"
Week 2 reality: "Also need file uploads, reactions,
                 threads, search, notifications..."
Week 4: Nothing ships

✅ Use Component Inventory:
  → P0 features only for V1
  → Ship V1
  → Then add P1 features
```

**Fix:** Write down V1 scope on Day 1. Stick to it.

**Pitfall #7: Ignoring Mini-Flux Warnings**

```
Mini-Flux: ⚠️ This API is deprecated
You: "Eh, probably fine"
→ 2 weeks later: API shuts down, app breaks

✅ Listen to warnings:
  → Research alternative
  → Build with non-deprecated API
  → App keeps working
```

**Fix:** Treat 0.4-0.6 confidence as "investigate before proceeding."

---

## 🛠️ TOOLS & TECHNOLOGIES

**Development Environment**

- **Code Editor:** VS Code + Claude Code extension
- **AI Assistants:** Claude Sonnet 4.5, MiniMax-M2, Grok
- **Version Control:** Git + GitHub
- **Terminal:** iTerm2 (Mac) or Windows Terminal

**The Cory Method Stack**

Planning:
- Excalidraw (diagrams)
- Notion or Markdown (documentation)
- Linear or GitHub Projects (task tracking)

Development:
- Docker + Docker Compose
- PostgreSQL (database)
- Redis (caching)
- FastAPI (Python backend)
- Next.js (React frontend)

Testing:
- pytest (Python)
- Jest (JavaScript)
- Playwright (E2E)

Deployment:
- OVH Cloud (hosting)
- GitHub Actions (CI/CD)
- Traefik (reverse proxy + SSL)

**Mini-Flux Stack**

Core:
- LangGraph (agent orchestration)
- MiniMax-M2 or Claude Sonnet 4.5 (LLM)
- LangChain (tool abstractions)

Verification Sources:
- Tavily API (web search)
- arXiv API (academic papers)
- GitHub API (code examples)
- Serper API (Google search)

Storage:
- PostgreSQL (verified facts)
- Redis (recent checks cache)
- Pinecone (vector search)

Frontend:
- Next.js (web interface)
- WebSocket (real-time updates)
- Recharts (confidence visualizations)

**Cost Breakdown (Monthly)**

```
Development Phase:
  - Claude API: $0-20 (low usage for planning)
  - MiniMax-M2: $0 (free tier until Nov 2025)
  - GitHub: $0 (free tier)
  Total: $0-20/month

Production Phase:
  - OVH VPS: $20-50/month
  - Domain: $12/year
  - Mini-Flux Tavily: $0-50/month (pay per search)
  - PostgreSQL: $0 (self-hosted)
  - Redis: $0 (self-hosted)
  Total: $25-100/month

Scale Phase (500+ users):
  - OVH Bigger VPS: $100-200/month
  - CDN: $20/month
  - Backups: $10/month
  - Monitoring: $20/month
  Total: $150-250/month
```

---

## 📚 RESOURCES

### Official Documentation

- **The Cory Method:** Your blog
- **Mini-Flux:** GitHub
- **Claude Code:** docs.claude.com/claude-code
- **MiniMax-M2:** platform.minimax.io

### Learning Resources

- **FastAPI Tutorial:** fastapi.tiangolo.com
- **Next.js Docs:** nextjs.org/docs
- **LangGraph Guide:** langchain-ai.github.io/langgraph
- **Docker Tutorial:** docs.docker.com/get-started

### Community

- **AetherPro Discord:** Join for support
- **r/CoryMethod:** Reddit community
- **Twitter:** @AetherProTech

### Example Projects

All examples use The Cory Method + Mini-Flux:

1. **AetherChat** - Team communication tool (this guide)
2. **AetherVox** - Voice-enabled AI assistant
3. **Aletheia** - Research truth verification bot
4. **AetherOS** - Universal AI agent runtime
5. **ClipSmart** - Viral video content platform

View all examples on GitHub

---

## 🤝 CONTRIBUTING

This methodology is open source because we believe in helping the AI community build better, faster, and more reliable software.

**How to Contribute**

**Share Your Experience:**

```
markdown

Used The Cory Method? Tell us:
- What you built
- Timeline (planned vs actual)
- What worked
- What was hard
- Your modifications to the method

Submit: https://github.com/aetherpro/cory-method/discussions
```

**Report Issues:**

```
markdown

Found a gap in the methodology?
- What phase had issues
- What would improve it
- Suggested additions

Submit: https://github.com/aetherpro/cory-method/issues
```

**Add Examples:**

markdown

Built something with The Cory Method + Mini-Flux?
- Share your project
- Document your process
- Help others learn

Submit: https://github.com/aetherpro/examples

**Recognition**

Contributors get:

- Credit in documentation
- Featured on AetherPro blog
- Access to private Discord channel
- Early access to new tools

---

## 📝 LICENSE

**The Cory Method + Mini-Flux Framework**



**License:** MIT



**Attribution:** If you use this methodology, a link back would be appreciated but is not required:

```
markdown

Built with [The Cory Method](https://aetherpro.com/cory-method)
```

---

## 🎯 FINAL THOUGHTS

**From Cory Gibson**

> I developed this methodology not from theoretical study but from practical necessity. As a solo founder transitioning from 15 years as a Master Electrician to building AI companies, I needed a repeatable process that actually worked.

> The electrical background taught me that complex systems need proper planning, that you test each component before moving on, and that documentation saves your future self hours of debugging.

> The Cory Method is just those principles applied to software development. And Mini-Flux emerged from being burned too many times by AI hallucinations that cost days of wasted work.

> This isn't academic theory. It's battle-tested methodology that shipped 10+ production applications in the last 6 months. It works because it's pragmatic, not perfect.

> Use it. Modify it. Share what you learn. Let's build better software, together.

**— Cory Gibson**
Founder & CEO, AetherPro Technologies LLC
November 2025

---

## 📞 CONTACT & SUPPORT

**Getting Help**

**Method Questions:**

- Discord: AetherPro Server
- Email: support@aetherpro.com

**Technical Issues:**

- GitHub Issues: Report a bug
- Stack Overflow: Tag `cory-method`

**Business Inquiries:**

- Email: cory@aetherpro.com
- Website: aetherpro.com

**Stay Updated**

- **Blog:** aetherpro.com/blog
- **Twitter:** @AetherProTech
- **Newsletter:** Subscribe

---

**VERSION HISTORY**

- v2.0 (November 2025) - Added Mini-Flux integration, expanded examples
- v1.5 (October 2025) - Added Phase 7 (Deployment), more code examples
- v1.0 (September 2025) - Initial public release
- v0.5 (August 2025) - Internal AetherPro framework

---

**END OF DOCUMENTATION**

*Now stop reading and go build something amazing.* 🚀