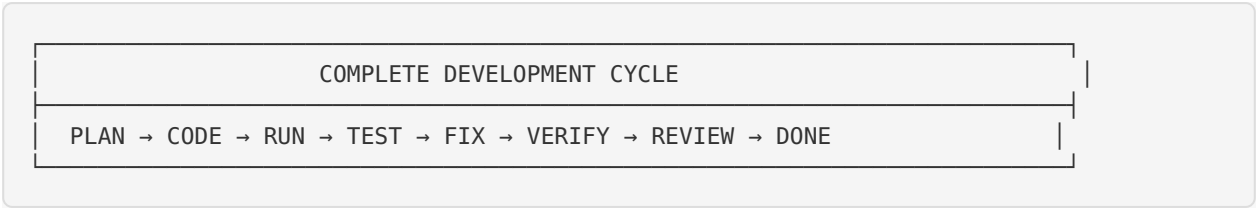
Complete Development Workflow Guide

This guide shows how the AI Orchestrator handles a complete development cycle: from planning to working, tested code. The orchestrator's execution, auto-fix, and verification loop capabilities make it a powerful assistant for full project development.

Table of Contents

Overview

The AI Orchestrator now supports a complete development workflow:

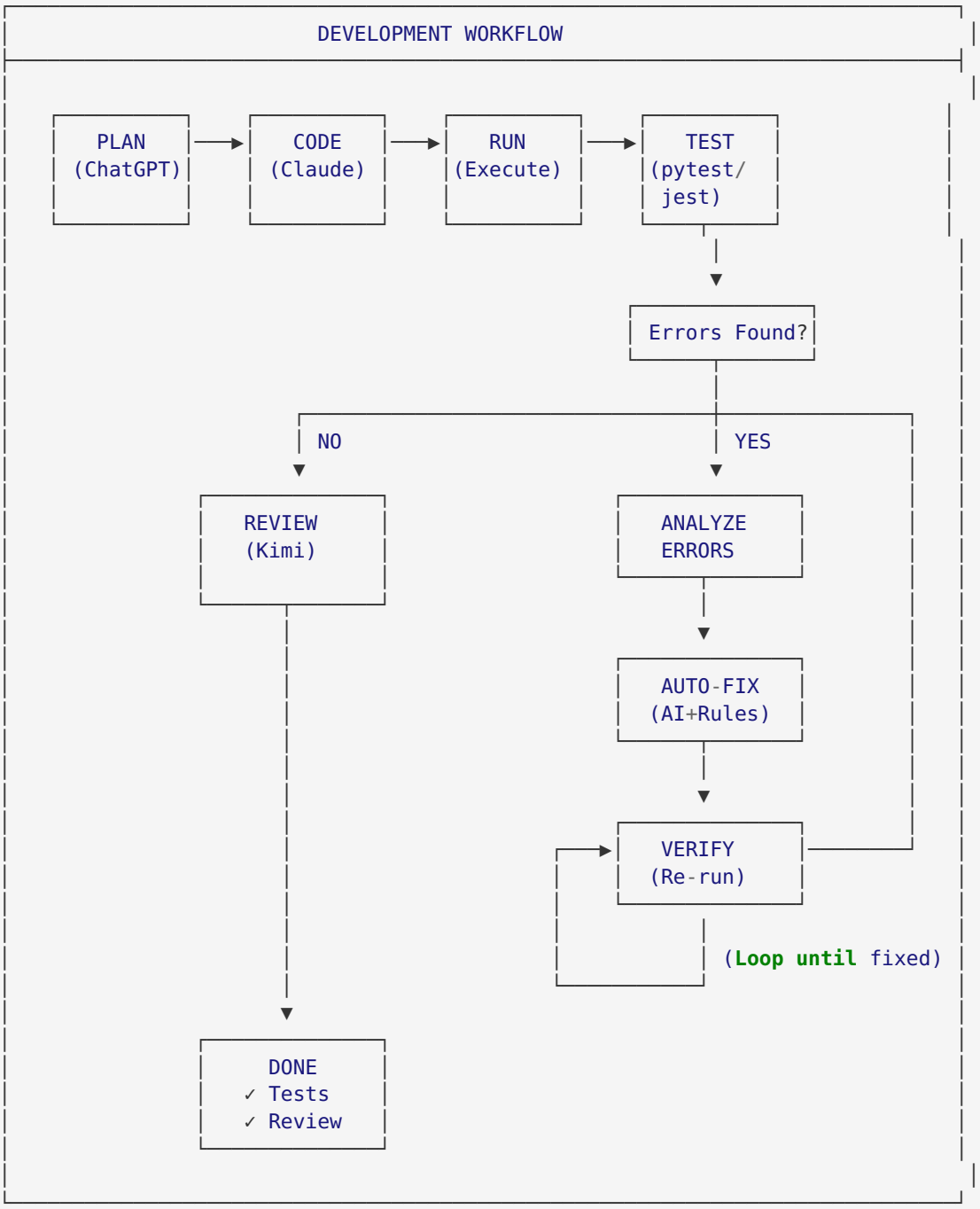


Key Capabilities

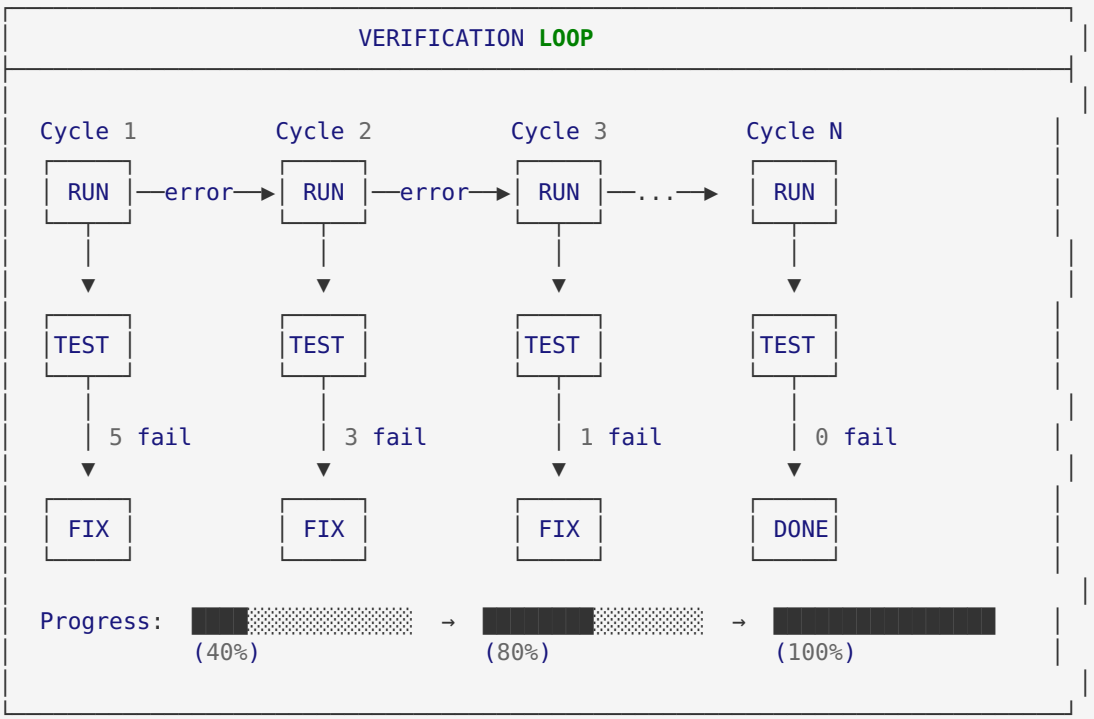
Phase	Tool	Purpose
Plan	`orchestrate_task`	Architecture & design
Code	`orchestrate_task`	Implementation
Run	`run_project`	Execute the project
Test	`test_project`	Run test suites
Fix	`fix_issues`	Auto-fix detected errors
Verify	`verify_project`	Loop until success
Review	`orchestrate_task`	Code quality review

Workflow Diagram

Full Development Cycle



Verification Loop Detail



Step-by-Step Development Guide

Phase 1: Planning (Architecture Design)

Start every project with proper planning. The orchestrator routes this to ChatGPT.

```
@ai-orchestrator orchestrate_task("Design the architecture for a REST API
that handles user authentication with JWT tokens, password reset via email,
and role-based access control")
```

What happens:
1. ChatGPT creates a high-level architecture
2. Defines API endpoints and data models
3. Outlines authentication flow
4. Suggests folder structure

Output includes:
- System architecture diagram
- Database schema
- API endpoint specifications
- Security considerations

Phase 2: Implementation

With the architecture in place, request the implementation. Claude handles this.

```
@ai-orchestrator orchestrate_task("Implement the user authentication system
based on the architecture. Include:
- User model with password hashing
- JWT token generation and validation
- Login, register, and password reset endpoints
- Middleware for protected routes")
```

What happens:
1. Claude generates the implementation code
2. Creates necessary files and modules
3. Implements error handling
4. Adds input validation

Phase 3: Execution

Now run the project to verify it starts correctly.

```
@ai-orchestrator run_project("/path/to/project")
```

What happens:
1. Auto-detects project type (Python, Node.js, etc.)
2. Installs dependencies if needed
3. Runs the project
4. Captures stdout/stderr
5. Reports any startup errors

Possible outcomes:
- ✅ **Success**: Project starts without errors
- ❌ **Error**: Dependency missing, syntax error, etc.

Phase 4: Testing

Run the test suite to verify functionality.

```
@ai-orchestrator test_project("/path/to/project")
```

What happens:
1. Detects test framework (pytest, jest, etc.)
2. Runs all tests
3. Parses results
4. Reports pass/fail counts

Phase 5: Error Analysis (If Needed)

When errors occur, analyze them:

```
@ai-orchestrator analyze_errors("/path/to/project")
```

Output includes:
- Error categorization (syntax, runtime, dependency, etc.)
- Root cause analysis

- File and line numbers
- Suggested fixes

Phase 6: Auto-Fix

Apply automated fixes to detected issues:

```
@ai-orchestrator fix_issues("/path/to/project")
```

What happens:
1. AI analyzes each error
2. Generates fix proposals
3. Validates fixes before applying
4. Creates backups
5. Applies fixes
6. Reports what was changed

Phase 7: Verification Loop

For complex issues, use the verification loop:

```
@ai-orchestrator verify_project("/path/to/project")
```

What happens:
1. Runs project and tests
2. If errors found, analyzes and fixes them
3. Re-runs to verify the fix worked
4. Repeats until all tests pass (or max cycles reached)
5. Generates comprehensive report

Phase 8: Code Review

Final review of the completed code:

```
@ai-orchestrator orchestrate_task("Review the authentication module for:
- Security vulnerabilities
- Best practices
- Performance issues
- Code quality")
```

The Verification Loop

The verification loop is the most powerful feature for fixing complex issues.

How It Works

```
┌─────────────────────────────────────────────────────────────┐
│                    VERIFICATION LOOP LOGIC                  │
├─────────────────────────────────────────────────────────────┤
│                                                             │
│  1. EXECUTE                                                 │
│     ├─ Run project                                          │
│     ├─ Capture output                                       │
│     └─ Check for errors                                     │
│                                                             │
│  2. TEST                                                    │
│     ├─ Run test suite                                       │
│     ├─ Parse results                                        │
│     └─ Count failures                                       │
│                                                             │
│  3. ANALYZE (if errors found)                               │
│     ├─ Categorize errors                                    │
│     ├─ Identify root causes                                 │
│     └─ Check for patterns                                   │
│                                                             │
│  4. FIX                                                     │
│     ├─ Generate fix using AI                                │
│     ├─ Validate fix safety                                  │
│     ├─ Backup original files                                │
│     └─ Apply fix                                            │
│                                                             │
│  5. TRACK PROGRESS                                          │
│     ├─ Compare to previous cycle                            │
│     ├─ Detect infinite loops                                │
│     └─ Decide: continue or stop                             │
│                                                             │
│  6. REPEAT or FINISH                                        │
│     ├─ Success → Generate report, done                      │
│     ├─ Progress → Continue loop                             │
│     └─ Stuck → Stop, recommend manual fix                   │
│                                                             │
└─────────────────────────────────────────────────────────────┘
```

Loop Protection

The orchestrator prevents infinite loops:

- **Max Cycles**: Default 10 cycles maximum
- **Same Error Detection**: Stops if same error occurs 3+ times
- **Progress Tracking**: Stops if no improvement after 3 cycles
- **Regression Detection**: Stops if fixes make things worse

Example Loop Output

```
╔═══════════════════════════════════════════════════════════╗
║                  VERIFICATION LOOP REPORT                 ║
╠═══════════════════════════════════════════════════════════╣
║ Status: SUCCESS                                           ║
║ Cycles Completed: 3                                       ║
║ Total Errors Fixed: 5                                     ║
╠═══════════════════════════════════════════════════════════╣
║ CYCLE 1:                                                  ║
║   ├─ Errors: 5 (3 dependency, 2 syntax)                   ║
║   ├─ Fixed: 3                                             ║
║   └─ Progress: ████░░░░░░░░░░░░░░░░ 20%                   ║
╠═══════════════════════════════════════════════════════════╣
║ CYCLE 2:                                                  ║
║   ├─ Errors: 2 (1 runtime, 1 import)                      ║
║   ├─ Fixed: 1                                             ║
║   └─ Progress: ████████████░░░░░░░░ 60%                   ║
╠═══════════════════════════════════════════════════════════╣
║ CYCLE 3:                                                  ║
║   ├─ Errors: 1 (1 logic)                                  ║
║   ├─ Fixed: 1                                             ║
║   └─ Progress: ████████████████████ 100%                  ║
╠═══════════════════════════════════════════════════════════╣
║ ✅ All tests passing                                      ║
╚═══════════════════════════════════════════════════════════╝
```

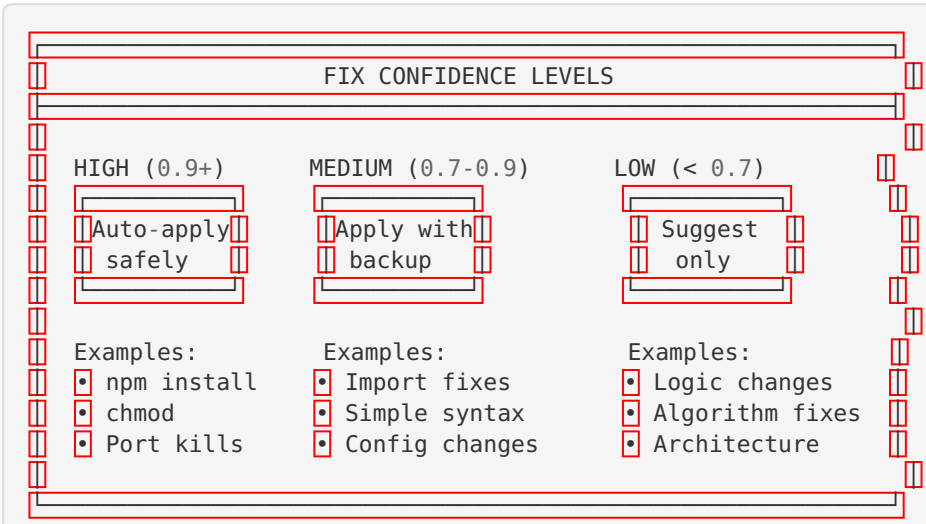
Auto-Fix in Action

Supported Fix Types

Error Type	Auto-Fix Method	Example
Missing dependency	Install command	`npm install express`
Syntax error	AI code correction	Fix missing bracket
Import error	Add missing import	`from flask import Flask`
Port conflict	Kill process or change port	`lsof -ti:3000 | xargs kill`
Permission denied	chmod command	`chmod +x script.sh`
Missing env var	Create from template	Copy `.env.example`
Type error	AI type annotation	Add proper types
Runtime error	AI analysis & fix	Logic correction

Fix Confidence Levels

Each fix has a confidence score:



Before/After Example

Error:

```
ModuleNotFoundError: No module named 'flask'
```

Auto-Fix Process:

```
1. DETECT:   Dependency error - missing 'flask'
2. STRATEGY: Install via pip
3. COMMAND:  pip install flask
4. VALIDATE: Check package exists
5. APPLY:    Run installation
6. VERIFY:   Re-run project
```

Result: ✅ Project now runs successfully

Best Practices

1. Start with Clear Requirements

```
✅ GOOD:
"Create a REST API with:
- GET /users - list all users
- POST /users - create user
- GET /users/:id - get specific user
Use Express.js, validate inputs, return JSON"

❌ BAD:
"Make an API for users"
```

2. Use the Full Workflow

Don’t skip phases:

```
✅ GOOD:
1. Plan architecture
2. Implement code
3. Run project
4. Run tests
5. Fix issues
6. Review

❌ BAD:
1. Write code
2. Hope it works
```

3. Check Verification Reports

Always review the verification report:

```
// Read the report to understand what was fixed
@ai-orchestrator verify_project("/path/to/project")

// Check the report output for:
// - What errors were found
// - How they were fixed
// - What still needs attention
```

4. Use Analyze Before Fix

For complex errors, analyze first:

```
// Step 1: Understand the problem
@ai-orchestrator analyze_errors("/path/to/project")

// Step 2: Review analysis
// ... read the categorization and suggestions ...

// Step 3: Then fix
@ai-orchestrator fix_issues("/path/to/project")
```

5. Backup Important Code

The orchestrator creates backups, but for critical code:

```
# Create your own backup before major changes
git commit -am "Before auto-fix"
```

6. Don’t Fix Infinitely

If the loop keeps failing:

```
Stop after 3-5 cycles of no progress

┌─────────────────────────────────────┐
│ Manual intervention likely needed for:  │
│ • Architecture problems                 │
│ • Business logic errors                 │
│ • Integration issues                    │
│ • External service problems             │
└─────────────────────────────────────┘
```

Common Scenarios

Scenario 1: New Project from Scratch

```
# 1. Design
@ai-orchestrator orchestrate_task("Design a blog API with posts, comments, and users")

# 2. Implement
@ai-orchestrator orchestrate_task("Implement the blog API using Flask")

# 3. Verify everything works
@ai-orchestrator verify_project("/path/to/blog")
```

Scenario 2: Fix Broken Tests

```
# Project exists but tests are failing
@ai-orchestrator verify_project("/path/to/project")

# Orchestrator will:
# 1. Run tests
# 2. Identify failures
# 3. Auto-fix test issues
# 4. Re-run until passing
```

Scenario 3: Debug Production Error

```
# 1. Analyze the error
@ai-orchestrator analyze_errors("/path/to/project",
  error_log="Error: ECONNREFUSED at line 42...")

# 2. Get fix suggestions
@ai-orchestrator fix_issues("/path/to/project")

# 3. Verify the fix
@ai-orchestrator verify_project("/path/to/project")
```

Scenario 4: Add Feature to Existing Project

```
# 1. Get the code reviewed first
@ai-orchestrator orchestrate_task("Review /path/to/project to understand structure")

# 2. Plan the addition
@ai-orchestrator orchestrate_task("Design how to add OAuth login to this project")

# 3. Implement
@ai-orchestrator orchestrate_task("Add OAuth login following the design")

# 4. Verify
@ai-orchestrator verify_project("/path/to/project")
```

Tool Quick Reference

Tool	Purpose	When to Use
`orchestrate_task`	Multi-model task execution	Planning, coding, reviewing
`run_project`	Execute project	Quick run check
`test_project`	Run tests	Verify functionality
`analyze_errors`	Understand errors	Before fixing complex issues
`fix_issues`	Auto-fix problems	After error analysis
`verify_project`	Full loop	Complete verification
`orches-trate_full_development`	Everything at once	New projects

Next Steps

1. **Try the examples**: See `examples/04_auto_fix_workflow.md`
2. **Full project example**: See `examples/05_full_development_cycle.md`
3. **Debugging guide**: See `examples/06_debugging_and_testing.md`
4. **Sample projects**: See `examples/sample_projects/`

Troubleshooting

Verification Loop Won’t Stop

Cause: Same error keeps occurring
Solution:
1. Check the error manually
2. Increase `max_same_error_attempts`
3. Or fix the root cause manually

Fixes Not Being Applied

Cause: Confidence too low
Solution:
1. Check fix confidence threshold
2. Lower `fix_confidence_threshold` in config
3. Or manually review and apply suggested fixes

Tests Always Fail

Cause: Tests may be wrong
Solution:

1. Analyze test code separately
2. Ensure tests are valid
3. Fix test issues before project issues