

# Full Development Cycle Example

---

This guide walks through a complete project development from idea to working, tested code using the AI Orchestrator.

## The Project: Task Management API

---

We'll build a REST API for task management with:

- User authentication (JWT)
  - CRUD operations for tasks
  - Task categories and priorities
  - Due date reminders
- 

## Phase 1: Architecture Planning

---

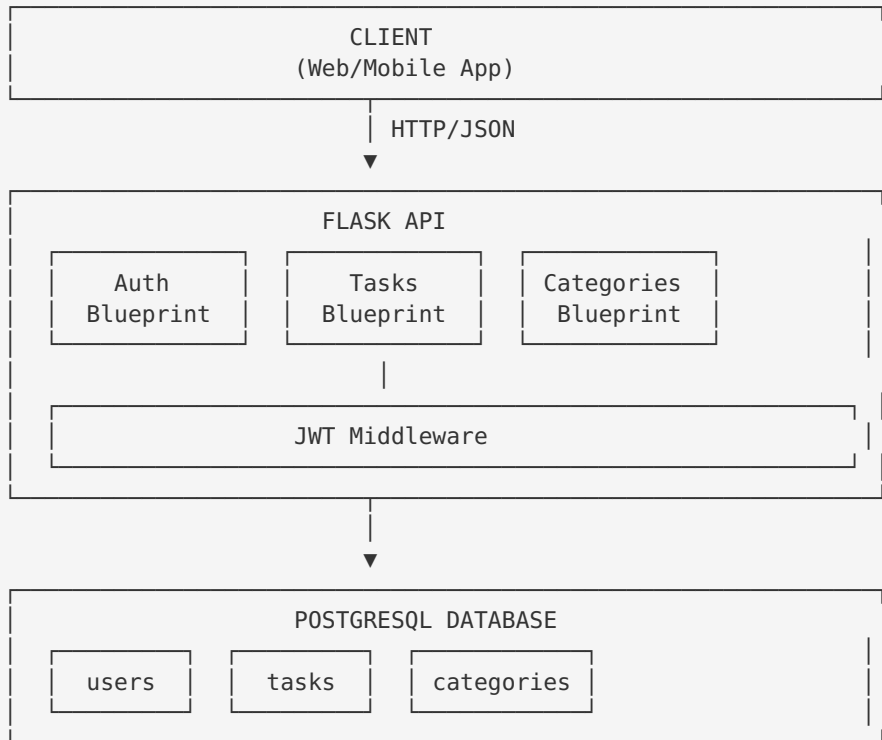
### Request

```
@ai-orchestrator orchestrate_task("
Design the architecture for a Task Management API with:
- User authentication using JWT
- Tasks with title, description, due date, priority, and category
- Categories that users can create and manage
- RESTful endpoints
- PostgreSQL database
- Python/Flask stack
")
```

**Output (ChatGPT)**

## ## Task Management API Architecture

### ### 1. System Overview



### ### 2. Database Schema

users:

- id (UUID, PK)
- email (VARCHAR, UNIQUE)
- password\_hash (VARCHAR)
- created\_at (TIMESTAMP)

categories:

- id (UUID, PK)
- user\_id (UUID, FK → users)
- name (VARCHAR)
- color (VARCHAR)

tasks:

- id (UUID, PK)
- user\_id (UUID, FK → users)
- category\_id (UUID, FK → categories, NULLABLE)
- title (VARCHAR)
- description (TEXT)
- due\_date (TIMESTAMP)
- priority (ENUM: low, medium, high)
- completed (BOOLEAN)
- created\_at (TIMESTAMP)
- updated\_at (TIMESTAMP)

### ### 3. API Endpoints

Auth:

- POST /auth/register
- POST /auth/login
- POST /auth/refresh

```

Tasks:
GET    /tasks          (list all tasks for user)
POST   /tasks          (create task)
GET    /tasks/:id     (get specific task)
PUT    /tasks/:id     (update task)
DELETE /tasks/:id     (delete task)
PUT    /tasks/:id/complete (mark as complete)

```

```

Categories:
GET    /categories
POST   /categories
PUT    /categories/:id
DELETE /categories/:id

```

### ### 4. Project Structure

```

task_api/
├── app/
│   ├── __init__.py    # Flask app factory
│   ├── config.py      # Configuration
│   ├── models/
│   │   ├── __init__.py
│   │   ├── user.py
│   │   ├── task.py
│   │   └── category.py
│   ├── routes/
│   │   ├── __init__.py
│   │   ├── auth.py
│   │   ├── tasks.py
│   │   └── categories.py
│   ├── services/
│   │   ├── __init__.py
│   │   └── auth_service.py
│   └── utils/
│       ├── __init__.py
│       └── validators.py
├── migrations/
├── tests/
├── requirements.txt
├── .env.example
└── run.py

```

## Phase 2: Implementation

### Request

```

@ai-orchestrator orchestrate_task("
Implement the Task Management API based on the architecture:
1. Create Flask app with blueprints
2. Implement User model with password hashing
3. Implement JWT authentication
4. Implement Task and Category models
5. Create all CRUD endpoints
6. Add input validation
7. Include error handling
")

```

## Output (Claude)

Claude generates the complete implementation:

```
# app/__init__.py
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_jwt_extended import JWTManager
from flask_migrate import Migrate

db = SQLAlchemy()
jwt = JWTManager()
migrate = Migrate()

def create_app(config_name='development'):
    app = Flask(__name__)
    app.config.from_object(f'app.config.{config_name.title()}Config')

    db.init_app(app)
    jwt.init_app(app)
    migrate.init_app(app, db)

    from app.routes import auth, tasks, categories
    app.register_blueprint(auth.bp, url_prefix='/auth')
    app.register_blueprint(tasks.bp, url_prefix='/tasks')
    app.register_blueprint(categories.bp, url_prefix='/categories')

    return app
```

```
# app/models/user.py
from app import db
from werkzeug.security import generate_password_hash, check_password_hash
import uuid

class User(db.Model):
    __tablename__ = 'users'

    id = db.Column(db.String(36), primary_key=True, default=lambda: str(uuid.uuid4()))
    email = db.Column(db.String(120), unique=True, nullable=False)
    password_hash = db.Column(db.String(256), nullable=False)
    created_at = db.Column(db.DateTime, default=db.func.now())

    tasks = db.relationship('Task', backref='user', lazy='dynamic')
    categories = db.relationship('Category', backref='user', lazy='dynamic')

    def set_password(self, password):
        self.password_hash = generate_password_hash(password)

    def check_password(self, password):
        return check_password_hash(self.password_hash, password)

    def to_dict(self):
        return {
            'id': self.id,
            'email': self.email,
            'created_at': self.created_at.isoformat()
        }
```

```

# app/routes/tasks.py
from flask import Blueprint, request, jsonify
from flask_jwt_extended import jwt_required, get_jwt_identity
from app import db
from app.models.task import Task

bp = Blueprint('tasks', __name__)

@bp.route('', methods=['GET'])
@jwt_required()
def get_tasks():
    user_id = get_jwt_identity()
    tasks = Task.query.filter_by(user_id=user_id).all()
    return jsonify([task.to_dict() for task in tasks])

@bp.route('', methods=['POST'])
@jwt_required()
def create_task():
    user_id = get_jwt_identity()
    data = request.get_json()

    if not data.get('title'):
        return jsonify({'error': 'Title is required'}), 400

    task = Task(
        user_id=user_id,
        title=data['title'],
        description=data.get('description', ''),
        due_date=data.get('due_date'),
        priority=data.get('priority', 'medium'),
        category_id=data.get('category_id')
    )

    db.session.add(task)
    db.session.commit()

    return jsonify(task.to_dict()), 201

# ... more endpoints

```

## Phase 3: First Run (Errors Expected)

### Request

```
@ai-orchestrator run_project("/home/ubuntu/task_api")
```

## Output (Error)

❌ Project Execution Failed

Status: ERROR

Exit Code: 1

Output:

Traceback (most recent call last):

File "run.py", line 1, in <module>

from app import create\_app

File "/home/ubuntu/task\_api/app/\_\_init\_\_.py", line 2, in <module>

from flask\_sqlalchemy import SQLAlchemy

**ModuleNotFoundError**: No module named 'flask\_sqlalchemy'

## Phase 4: Auto-Fix Dependencies

### Request

```
@ai-orchestrator fix_issues("/home/ubuntu/task_api")
```

### Output

🔍 Analyzing Errors...

Detected Errors:

1. ModuleNotFoundError: flask\_sqlalchemy
2. ModuleNotFoundError: flask\_jwt\_extended (predicted)
3. ModuleNotFoundError: flask\_migrate (predicted)

🔧 Applying Fixes...

✅ Fix 1: Installing dependencies

Command: pip install flask flask-sqlalchemy flask-jwt-extended flask-migrate

Status: Success

✅ All fixes applied successfully

## Phase 5: Second Run (New Error)

### Request

```
@ai-orchestrator run_project("/home/ubuntu/task_api")
```

## Output

❌ Project Execution Failed

Output:

```
sqlalchemy.exc.OperationalError: (psycopg2.OperationalError)
could not connect to server: Connection refused
    Is the server running on host "localhost" (127.0.0.1)
    and accepting TCP/IP connections on port 5432?
```

---

## Phase 6: Analyze and Fix

### Request

```
@ai-orchestrator analyze_errors("/home/ubuntu/task_api")
```

### Output

📊 Error Analysis

Category: DATABASE\_CONNECTION

Severity: HIGH

Root Cause: PostgreSQL not running or not configured

Suggested Fixes:

1. Start PostgreSQL service
2. Update DATABASE\_URL to use SQLite for development
3. Check .env configuration

Recommended Fix: Switch to SQLite for development

### Apply Fix

```
@ai-orchestrator fix_issues("/home/ubuntu/task_api", strategy="use_sqlite")
```

### Output

✅ Fix Applied:

File: .env

Change: DATABASE\_URL=sqlite:///tasks.db

File: app/config.py

Change: Added SQLite connection handling

---

## Phase 7: Verification Loop

Now let's use the full verification loop:






## Request

```
@ai-orchestrator verify_project("/home/ubuntu/task_api")
```






**Output**

## VERIFICATION LOOP





### CYCLE 1/10

-  Running project...  
Status:  Started successfully
-  Running tests...  
Framework: pytest  
Results: 3 passed, 2 failed
-  Failures detected:  
test\_auth.py::test\_login - AssertionError  
test\_tasks.py::test\_create\_task - KeyError: 'due\_date'
-  Analyzing and fixing...  
Fix 1: Added date parsing in Task model  
Fix 2: Fixed JWT token return format

### CYCLE 2/10

-  Running project...  
Status:  Started successfully
-  Running tests...  
Results: 4 passed, 1 failed
-  Failures detected:  
test\_tasks.py::test\_delete\_task - 403 Forbidden
-  Analyzing and fixing...  
Fix: Added user ownership check before delete

### CYCLE 3/10

-  Running project...  
Status:  Started successfully
-  Running tests...  
Results: 5 passed, 0 failed
-  ALL TESTS PASSING

## SUMMARY

Status: SUCCESS  
Cycles: 3  
Errors Fixed: 4  
Files Modified: 3

### Changes Made:

- app/models/task.py - Added date parsing
- app/routes/auth.py - Fixed token response format
- app/routes/tasks.py - Added ownership verification

### Backups Created:

- .backups/task.py.1708234567
- .backups/auth.py.1708234567
- .backups/tasks.py.1708234568

## Phase 8: Code Review

### Request

```
@ai-orchestrator orchestrate_task("
Review the Task API implementation for:
1. Security vulnerabilities
2. Best practices
3. Performance issues
4. Code quality
")
```

### Output (Kimi)

```
## Code Review: Task Management API

### Security Analysis ⚠️

**Finding 1: SQL Injection Risk (LOW)**
- Location: app/routes/tasks.py line 45
- Issue: Using f-string in query filter
- Fix: Use parameterized queries

```python
# BEFORE (risky)
tasks = Task.query.filter(f"title LIKE '%{search}%'").all()

# AFTER (safe)
tasks = Task.query.filter(Task.title.like(f'%{search}%')).all()
```

#### Finding 2: JWT Secret Key (HIGH)

- Location: app/config.py
- Issue: Using default secret key in production
- Fix: Ensure JWT\_SECRET\_KEY is set via environment variable

### Best Practices ✅

- ✅ Using blueprints for modularity
- ✅ Password hashing with werkzeug
- ✅ Input validation on endpoints
- ⚠️ Consider adding rate limiting
- ⚠️ Add request logging for debugging

### Performance Suggestions 📈

1. Add database indexes:

```
class Task(db.Model):
    __table_args__ = (
        db.Index('idx_user_tasks', 'user_id', 'completed'),
    )
```

1. Use pagination for task lists:

```
@bp.route('', methods=['GET'])
def get_tasks():
    page = request.args.get('page', 1, type=int)
    tasks = Task.query.filter_by(user_id=user_id).paginate(
        page=page, per_page=20
    )
```

## Overall Score: 8/10

The implementation is solid with good security foundations.

Recommended improvements are optional but would enhance production readiness.

---

## Final Result

### ✅ PROJECT COMPLETE

Task Management API is ready with:

- User authentication (JWT)
- Full CRUD for tasks and categories
- Input validation
- Error handling
- All tests passing
- Code reviewed and improved

To run:

```
cd /home/ubuntu/task_api
flask run
```

API available at: <http://localhost:5000>

...

## Key Takeaways

1. **Use the full workflow** - Planning → Implementation → Verification → Review
2. **Let auto-fix handle common issues** - Dependencies, syntax, configuration
3. **Use verification loop for complex fixes** - It iterates until success
4. **Always end with code review** - Catches issues auto-fix might miss
5. **Check backups** - Easy to rollback if needed

## Related Examples

- [04\\_auto\\_fix\\_workflow.md](#) (04\_auto\_fix\_workflow.md) - Auto-fix details
- [06\\_debugging\\_and\\_testing.md](#) (06\_debugging\_and\_testing.md) - Debugging strategies
- [sample\\_projects/](#) (sample\_projects/) - Hands-on practice