Example 3: Code Review and Refactoring

This example demonstrates how to use the AI Orchestrator for comprehensive code review workflows, including security analysis, performance optimization, and refactoring.

Workflow Overview

```
┌─────────────────────────────────────────────────────────────┐
│                     CODE REVIEW WORKFLOW                      │
└─────────────────────────────────────────────────────────────┘
                              │
        ┌─────────────────────┼─────────────────────┐
        │                     │                     │
        ▼                     ▼                     ▼
┌───────────────┐   ┌───────────────┐   ┌───────────────┐
│   SECURITY    │   │  PERFORMANCE  │   │     STYLE     │
│    (Kimi)     │   │   (Gemini)    │   │   (Claude)    │
│               │   │               │   │               │
│ - Vulns       │   │ - Complexity  │   │ - Patterns    │
│ - Injection   │   │ - Memory      │   │ - Readability│
│ - Auth        │   │ - Algorithms  │   │ - Standards   │
└───────┬───────┘   └───────┬───────┘   └───────┬───────┘
        │                   │                   │
        └───────────────────┼───────────────────┘
                            │
                            ▼
            ┌───────────────────────┐
            │  CONSOLIDATE          │
            │  FINDINGS             │
            └───────────┬───────────┘
                        │
                        ▼
            ┌───────────────────────┐
            │  IMPLEMENT FIXES      │
            │  (Claude)             │
            └───────────────────────┘
```

Scenario: Review a Python API Module

Let’s review a user authentication module that needs comprehensive review.

Sample Code to Review

```python
# auth.py - User authentication module
import jwt
import hashlib
from datetime import datetime, timedelta
from typing import Optional
from fastapi import HTTPException, Depends
from fastapi.security import HTTPBearer
from sqlalchemy.orm import Session
from database import get_db
from models import User

SECRET_KEY = "mysecretkey123"
ALGORITHM = "HS256"

security = HTTPBearer()

def hash_password(password: str) -> str:
    return hashlib.md5(password.encode()).hexdigest()

def verify_password(plain_password: str, hashed_password: str) -> bool:
    return hash_password(plain_password) == hashed_password

def create_token(user_id: int) -> str:
    expire = datetime.utcnow() + timedelta(days=30)
    payload = {"user_id": user_id, "exp": expire}
    return jwt.encode(payload, SECRET_KEY, algorithm=ALGORITHM)

def get_current_user(token = Depends(security), db: Session = Depends(get_db)) ->
User:
    try:
        payload = jwt.decode(token.credentials, SECRET_KEY, algorithms=[ALGORITHM])
        user_id = payload.get("user_id")
        user = db.query(User).filter(User.id == user_id).first()
        return user
    except:
        raise HTTPException(status_code=401, detail="Invalid token")

def login(email: str, password: str, db: Session) -> dict:
    user = db.query(User).filter(User.email == email).first()
    if not user:
        raise HTTPException(status_code=401, detail="User not found")
    if not verify_password(password, user.password):
        raise HTTPException(status_code=401, detail="Wrong password")
    token = create_token(user.id)
    return {"access_token": token, "token_type": "bearer"}

def register(email: str, password: str, name: str, db: Session) -> User:
    existing = db.query(User).filter(User.email == email).first()
    if existing:
        raise HTTPException(status_code=400, detail="Email already registered")
    user = User(email=email, password=hash_password(password), name=name)
    db.add(user)
    db.commit()
    return user
```

Step 1: Security Review (Kimi)

~~~
@ai-orchestrator route_to_model("
Perform a comprehensive security review of this authentication module:

```python
[Paste the auth.py code above]
~~~

Review for:
1. Password hashing security
2. JWT implementation vulnerabilities
3. Secret key management
4. Error handling and information leakage
5. Input validation
6. Session/token management
7. SQL injection risks
8. Timing attack vulnerabilities
9. Rate limiting needs
10. OWASP Top 10 compliance

For each issue found:
- Severity: Critical/High/Medium/Low
- Description of the vulnerability
- Potential attack vector
- Recommended fix

", “moonshot”)

```
### Expected Security Findings

| Issue | Severity | Description |
|-------|----------|-------------|
| MD5 password hashing | **CRITICAL** | MD5 is cryptographically broken |
| Hardcoded secret key | **CRITICAL** | Secret exposed in source code |
| 30-day token expiration | **HIGH** | Too long, no refresh mechanism |
| Generic exception handling | **MEDIUM** | Hides actual errors |
| User enumeration | **MEDIUM** | Different messages for user/password |
| No password requirements | **MEDIUM** | Weak passwords allowed |
| No rate limiting | **MEDIUM** | Brute force possible |

---

## Step 2: Performance Analysis (Gemini)
```

@ai-orchestrator route_to_model(“
Analyze the performance characteristics of this authentication module:

```
[Paste the auth.py code]
```

Analyze:
1. Database query efficiency
2. Password hashing performance implications

3. Token generation/validation overhead
4. Memory usage patterns
5. Potential bottlenecks under load
6. Scalability considerations

Provide:
- Performance metrics to monitor
- Optimization recommendations
- Load testing suggestions

”, “gemini”)

```
---

## Step 3: Code Style & Best Practices (Claude)
```

@ai-orchestrator route_to_model(“
Review this authentication code for Python best practices and style:

```
[Paste the auth.py code]
```

Review for:
1. PEP 8 compliance
2. Type hints completeness
3. Documentation/docstrings
4. Error handling patterns
5. Code organization
6. Naming conventions
7. SOLID principles adherence
8. Testability

Provide specific suggestions with code examples.

”, “anthropic”)

```
---

## Step 4: Implement Security Fixes (Claude)

Based on the security review findings:
```

@ai-orchestrator route_to_model(“
Refactor the authentication module to fix the security vulnerabilities identified:

Issues to fix:
1. Replace MD5 with bcrypt for password hashing
2. Move SECRET_KEY to environment variables
3. Implement proper token expiration with refresh tokens
4. Add password strength requirements
5. Fix user enumeration (generic error messages)
6. Add proper exception handling
7. Add type hints and documentation

Original code:

```
[Paste auth.py]
```

Requirements:
- Use bcrypt for passwords
- Use Pydantic Settings for configuration
- Implement access/refresh token pair
- Add password validation
- Maintain backward compatibility where possible
- Add comprehensive docstrings

", "anthropic")

```
---

## Step 5: Review the Refactored Code (Kimi)

After Claude implements the fixes:
```

@ai-orchestrator route_to_model("
Review the refactored authentication module:

```
[Paste refactored code]
```

Verify that the following issues have been addressed:
1. ✅ MD5 replaced with bcrypt
2. ✅ Secret key from environment
3. ✅ Shorter token expiration
4. ✅ Refresh token mechanism
5. ✅ Generic error messages
6. ✅ Password requirements
7. ✅ Proper exception handling

Check for any new issues introduced during refactoring.
Confirm the security posture improvement.

", "moonshot")

~~~
---

## Expected Refactored Code

```python
# auth.py - Secure User Authentication Module
"""
Secure authentication module using bcrypt and JWT.

This module provides:
- Password hashing with bcrypt
- JWT access and refresh tokens
- Secure token validation
- Password strength requirements
"""

import re
from datetime import datetime, timedelta
from typing import Optional, Tuple

import bcrypt
import jwt
from fastapi import HTTPException, Depends, status
from fastapi.security import HTTPBearer
from pydantic import BaseSettings, validator
from sqlalchemy.orm import Session

from database import get_db
from models import User


class AuthSettings(BaseSettings):
    """Authentication configuration from environment."""

    secret_key: str
    algorithm: str = "HS256"
    access_token_expire_minutes: int = 30
    refresh_token_expire_days: int = 7
    min_password_length: int = 8

    class Config:
        env_prefix = "AUTH_"


settings = AuthSettings()
security = HTTPBearer()


class PasswordValidator:
    """Validate password strength requirements."""

    @staticmethod
    def validate(password: str) -> Tuple[bool, str]:
        """Check password meets security requirements."""
        if len(password) < settings.min_password_length:
            return False, f"Password must be at least {settings.min_password_length} characters"
        if not re.search(r"[A-Z]", password):
            return False, "Password must contain at least one uppercase letter"
        if not re.search(r"[a-z]", password):
            return False, "Password must contain at least one lowercase letter"
        if not re.search(r"\d", password):
~~~

```
            return False, "Password must contain at least one digit"
        return True, ""


def hash_password(password: str) -> str:
    """Hash password using bcrypt."""
    salt = bcrypt.gensalt()
    return bcrypt.hashpw(password.encode(), salt).decode()


def verify_password(plain_password: str, hashed_password: str) -> bool:
    """Verify password against bcrypt hash."""
    return bcrypt.checkpw(
        plain_password.encode(),
        hashed_password.encode()
    )


def create_access_token(user_id: int) -> str:
    """Create short-lived access token."""
    expire = datetime.utcnow() + timedelta(
        minutes=settings.access_token_expire_minutes
    )
    payload = {
        "user_id": user_id,
        "exp": expire,
        "type": "access"
    }
    return jwt.encode(payload, settings.secret_key, algorithm=settings.algorithm)


def create_refresh_token(user_id: int) -> str:
    """Create long-lived refresh token."""
    expire = datetime.utcnow() + timedelta(
        days=settings.refresh_token_expire_days
    )
    payload = {
        "user_id": user_id,
        "exp": expire,
        "type": "refresh"
    }
    return jwt.encode(payload, settings.secret_key, algorithm=settings.algorithm)


def get_current_user(
    token = Depends(security),
    db: Session = Depends(get_db)
) -> User:
    """Extract and validate current user from JWT token."""
    credentials_exception = HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED,
        detail="Could not validate credentials",
        headers={"WWW-Authenticate": "Bearer"},
    )

    try:
        payload = jwt.decode(
            token.credentials,
            settings.secret_key,
            algorithms=[settings.algorithm]
        )

        if payload.get("type") != "access":
            raise credentials_exception

        user_id: Optional[int] = payload.get("user_id")
        if user_id is None:
            raise credentials_exception

    except jwt.ExpiredSignatureError:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Token has expired",
            headers={"WWW-Authenticate": "Bearer"},
        )
    except jwt.JWTError:
        raise credentials_exception

    user = db.query(User).filter(User.id == user_id).first()
    if user is None:
        raise credentials_exception

    return user


def login(email: str, password: str, db: Session) -> dict:
    """Authenticate user and return token pair."""
    # Generic error message to prevent user enumeration
    invalid_credentials = HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED,
        detail="Invalid email or password"
    )

    user = db.query(User).filter(User.email == email).first()
    if not user:
        raise invalid_credentials

    if not verify_password(password, user.password):
        raise invalid_credentials

    return {
        "access_token": create_access_token(user.id),
        "refresh_token": create_refresh_token(user.id),
        "token_type": "bearer"
    }


def register(
    email: str,
    password: str,
    name: str,
    db: Session
) -> User:
    """Register a new user with password validation."""
    # Validate password strength
    is_valid, error_message = PasswordValidator.validate(password)
    if not is_valid:
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail=error_message
        )

    # Check for existing user
    existing = db.query(User).filter(User.email == email).first()
    if existing:
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail="Email already registered"
        )

    # Create new user
    user = User(
        email=email,
        password=hash_password(password),
        name=name
    )
    db.add(user)
    db.commit()
    db.refresh(user)

    return user
```

Review Workflow Summary

Step	Tool	Model	Purpose
1	route_to_model	Kimi	Security vulnerability scan
2	route_to_model	Gemini	Performance analysis
3	route_to_model	Claude	Style/best practices
4	route_to_model	Claude	Implement fixes
5	route_to_model	Kimi	Verify fixes

Quick Review Templates

Security-Only Review

```
@ai-orchestrator route_to_model("Security review: [paste code]", "moonshot")
```

Performance-Only Review

```
@ai-orchestrator route_to_model("Performance analysis: [paste code]", "gemini")
```

Full Review (Auto-Routed)

```
@ai-orchestrator orchestrate_task("Review this code for security, performance, and
best practices: [paste code]")
```

Best Practices for Code Review

1. **Be Specific**: Mention the language, framework, and review focus
2. **Provide Context**: Explain what the code does and its requirements
3. **Iterate**: Review → Fix → Review again
4. **Prioritize**: Address Critical/High issues first
5. **Document**: Keep track of review findings and fixes