Module 4: Intelligent Refactoring & CI/CD, Capstone Demonstration

Capstone: From Legacy to Lit-Up — Diagnose, Refactor, Test, and Ship a Real Codebase

Overview

This capstone proves the four claims that anchor Module 4 — and it does it on real, ugly, production-shaped code:

- AI can diagnose a messy file **faster and more thoroughly** than any human reviewer — code smells, complexity scores, priority matrix, all in under five minutes.
- Refactoring with AI is safe **when you constrain it** — Strangler Fig, prompt chaining, behavioral equivalence tests. Reckless rewrites are a discipline problem, not an AI problem.
- A test suite is no longer a chore. **You will scaffold one in minutes** — happy paths, boundary values, error states — and watch coverage climb from 0 to 80%+ live.
- A green CI/CD pipeline is **one prompt away** — lint, test, security, coverage gate, and an approval-gated deploy, all generated, pushed, and running on GitHub Actions in under ten minutes.

You will start with a deliberately messy ~100-line Python service (order_manager.py), run a full diagnosis, extract one God-Class responsibility into a clean service, generate a comprehensive test suite, and ship a production-grade GitHub Actions pipeline — all inside one 25-minute live session.

Before You Start

- Have Python 3.11+ and Git installed. Verify with python3 --version and git --version.
- Have an empty GitHub repository ready to receive a push (named something like module-4-capstone-demo).
- Have your AI editor open (Cursor or VS Code with Copilot/Continue) and Claude.ai open in a browser tab as a fallback.
- Install pytest, pytest-cov, radon, and flake8 ahead of time: pip install pytest pytest-cov radon flake8.
- Pre-clone the course repo with the legacy file order_manager.py at the project root. The full source is reproduced at the end of this document for reference.
- Open a terminal in the project directory before you start. Live demo time is precious — the prep is not.

During Hands-On — Four Segments at a Glance

- **SEGMENT 1** — Diagnose order_manager.py — health report, complexity score, priority matrix — ~5 min
- **SEGMENT 2** — Refactor the highest-priority smell using Strangler Fig and prompt chaining — ~7 min

- **SEGMENT 3** — Generate a full test suite — happy path, boundaries, errors — and check coverage — ~6 min
- **SEGMENT 4** — Generate a GitHub Actions pipeline, push, watch it go green, debug a failure live — ~7 min

Remember

Module 4 is not about writing new code. It is about taking code you would normally avoid touching and making it safe to evolve. Every prompt, every test, every YAML line in this demo exists for one reason: to drop the cost of changing a codebase to near-zero. By the end of these 25 minutes you will not just have refactored a file — you will have demonstrated the entire AI-native engineering workflow that the rest of the students will use for the rest of their careers.

Segment 1 — Diagnose the Patient

What to Learn

You cannot refactor what you have not measured. Before you write a single line of new code, AI gives you three things in under five minutes: a structured health report listing every code smell, a cyclomatic complexity score for each function, and a priority matrix telling you what to fix first. Skip diagnosis and you will refactor the wrong thing — beautifully.

Step 1. Generate the health report.

Open order_manager.py in your editor. Open the AI chat panel. Paste the entire file into the chat, then run this prompt:

```
You are a senior software architect.

Analyze the attached Python file. For each code smell you find, report:
  - The smell name (Long Method, God Class, Primitive Obsession, etc.)
  - The specific line numbers affected
  - A risk level (low / medium / high)
  - A one-sentence recommended fix

Format the response as a markdown table with these exact columns:
| Smell | Lines | Risk | Recommended Fix |
```

What you should get

A markdown table with roughly 6–8 rows. Expect: God Class (the entire OrderManager class), Long Method (process_order, ~60 lines, doing five things), Primitive Obsession (raw dicts everywhere instead of Order/LineItem objects), Duplicated Logic (the credit_card and paypal branches share 80% of their code), and a high-risk State Mutation Before Failure bug — payment is charged before

inventory is checked, so an out-of-stock item leaves a customer charged with no order. That last one is the one that pays for this entire course.

Step 2. Add the complexity score.

Drop into the terminal and run radon for an objective measurement:

```
radon cc order_manager.py -s -a
```

Radon will rate each function A through F. Expect process_order to come back as a D or E — somewhere around 12–15 cyclomatic complexity, well into "refactor this" territory. Then ask AI to confirm and explain:

```
Estimate the cyclomatic complexity of process_order in this file.
Explain which conditionals are driving the score the most.
Compare your estimate to what radon would report.
```

What you should get

Two numbers within a point of each other and a clear narrative: "the nested if for payment_method, combined with the inventory loop and the success/failure branch, is responsible for most of the complexity." The objective number plus the AI explanation gives the room a shared vocabulary for the next segment.

Step 3. Build the priority matrix.

Get change frequency from git so you know which problems are most active:

```
git log --pretty=format: --name-only | sort | uniq -c | sort -rn | head -10
```

Then ask AI to combine the health report with the change frequency data:

```
Combine the health report above with this change frequency data.
Produce a 2x2 priority matrix:
  - Axis 1: change frequency (high/low)
  - Axis 2: smell severity (high/low)
For each quadrant, list the smells that fall there and recommend an action:
  fix now, schedule, monitor, or leave alone.
```

What you should get

A clean four-quadrant breakdown. The State Mutation bug and the Long Method process_order should land in the "fix now" quadrant — high frequency, high severity. The God Class as a whole sits in "schedule." Anything cosmetic ends up in "leave alone." This is the artifact you will paste into your team's ticket tracker on Monday morning.

Segment 2 — Refactor with Strangler Fig and Prompt Chaining

What to Learn

Refactoring is **not** rewriting. Refactoring preserves behavior; rewriting is high-risk theater. The Strangler Fig pattern lets us grow a clean PaymentProcessor service alongside the legacy OrderManager, route to it incrementally, and retire the old code only when the new one is fully proven. Prompt chaining keeps each AI step small enough to review and reverse. Four prompts. Four reviewable diffs. Zero broken builds.

Step 1. List the responsibilities (prompt 1 of 4).

Tell AI exactly what we are about to extract — no more, no less:

```
List every distinct business responsibility in the OrderManager class.
For each, name it and identify the methods or code blocks that implement it.
Do not propose any changes yet. Just produce the inventory.
```

What you should get

A clean list: payment processing, inventory management, customer notification, reporting, and a logging cross-concern. Five domains in one class. That is the God Class diagnosis confirmed at the responsibility level. We are picking ONE — payment processing — for the live demo. The others become PRs for the students to do as homework.

Step 2. Extract PaymentProcessor (prompt 2 of 4).

This prompt is where most engineers go wrong by saying "rewrite the file." We do the opposite: we ask for an additive change, with explicit constraints.

```
Extract all payment-processing logic from OrderManager into a new class
called PaymentProcessor in a new file payment_processor.py. Constraints:

  - Use Clean Architecture conventions (dependency injection,
    no direct imports of infrastructure).
  - Preserve every method signature exactly.
  - Do NOT modify order_manager.py in this step. Generate the new
    file only.
  - Output as a unified diff with an inline comment on every changed
    block explaining the architectural intent.
```

What you should get

A new file payment_processor.py with a focused PaymentProcessor class — constructor takes a payment_gateway and a logger, exposes a charge(amount, payment_method, payment_details) method returning a typed result. Comments next to each block explain why it exists. Notice the AI did not touch the original file. That is the Strangler Fig discipline: grow the new, do not bulldoze the old.

Step 3. Wire OrderManager to delegate (prompt 3 of 4).

Now we route the host through the fig. This prompt explicitly limits the diff to the changed methods only — no whole-file rewrites:

```
Update OrderManager to delegate all payment-related work to a
PaymentProcessor instance injected via the constructor. Constraints:

  - Show ONLY the changed methods, not the whole file.
  - The public interface of OrderManager must remain byte-for-byte
    identical for callers.
  - Every change must have an inline comment explaining the delegation.
```

What you should get

A small, surgical diff: the constructor now takes a payment_processor parameter, and the payment branches inside process_order are replaced with self.payment_processor.charge(...). Five to ten changed lines. The diff is small enough to read in one breath, which is exactly the point.

Step 4. Generate behavioral equivalence tests (prompt 4 of 4).

This is the safety net. We do not merge until the new code provably behaves like the old code:

```
Generate a pytest test suite that verifies the refactored OrderManager
behaves identically to the pre-refactor version for the payment flow.

Constraints:
  - Use unittest.mock for every external dependency (db, email_client,
    payment_gateway).
  - Cover: successful credit_card charge, successful paypal charge,
    declined card, payment gateway timeout, and an invalid amount.
  - Do NOT assert correctness — assert consistency with the documented
    pre-refactor outputs given in the comment block of process_order.
  - Name every test using the pattern:
    function_when_condition_then_expected_result
```

What you should get

Five tests, each named like process_order_when_credit_card_charge_succeeds_then_returns_ok and each with a one-line docstring explaining what behavior would break in production if it failed. Run pytest. All five go green. That green output is your license to merge — not the engineer's opinion, not a code review vibe, but a reproducible proof that the new code does what the old code did.

Segment 3 — Zero-Friction Test Generation

What to Learn

A strong test prompt has five components: role, context, framework, coverage targets, and an output format that includes a one-line failure explanation per test. Skip any of these and you will spend twenty minutes hand-fixing AI output. Include all five and the tests are PR-ready on the first run. We will generate the happy path, then boundary values (where AI consistently outperforms manual testing), then error states.

Step 1. Happy path first.

Always anchor the suite with a green test that documents correct behavior. The activation energy here is low — that is the point.

```
You are a senior QA engineer specializing in Python testing with pytest.

Generate a happy-path unit test for PaymentProcessor.charge with these inputs:
  amount = 49.99
  payment_method = "credit_card"
  payment_details = { "card_number": "4111111111111111", "cvv": "123" }

Expected output: a result object with status="success" and a non-empty
transaction_id field.

Mock the payment_gateway and logger using unittest.mock.
Use Arrange-Act-Assert structure.
Add a one-line comment explaining what failure this test would catch.
```

What you should get

A clean, runnable test in about 15 lines. Mocks pre-configured at the top. A comment that says something like "catches: PaymentProcessor returning success without a real transaction_id." Run pytest, watch it pass. That green is your behavioral baseline for everything that follows.

Step 2. Boundary value analysis.

This is where AI outperforms most human testers. It will find boundaries you would have missed:

```
Identify all boundary values for the inputs of PaymentProcessor.charge.

Generate one pytest test for each boundary, including values just inside
and just outside each limit. Cover at minimum:
  - amount: 0, 0.01, 0.99, 1.00, max float
  - amount: negative values
  - card_number: empty string, 15 digits, 16 digits, 17 digits
  - cvv: 2 digits, 3 digits, 4 digits

Use the naming pattern:
  charge_when_<condition>_then_<expected_result>
```

What you should get

Roughly a dozen tests. Each one targets a specific boundary. Watch the room when they see the test count — most engineers in the audience would have written three or four boundaries by hand. AI finds all of them, every time. That is the multiplier the entire course is built on.

Step 3. Error states (the ones that bite in production).

The category of tests almost no one writes until production burns. Mock failures, assert exception types, and verify no state mutation occurred before the failure:

```
Generate pytest tests for every error state PaymentProcessor.charge can reach:
  - payment_gateway raises ConnectionError
  - payment_gateway raises TimeoutError
  - payment_gateway returns { status: "declined" }
  - invalid card number format
  - amount above the gateway's daily limit

For each test:
  - Mock all external dependencies.
  - Assert on the specific exception type and message.
  - Verify that no logging, no notification, and no state mutation
    occurred before the failure.
```

What you should get

Five more tests, each one defending against a specific production failure mode. The "no state mutation before failure" assertion is the showstopper — that is the test that would have caught the State Mutation bug we found in Segment 1. Point at it. Tell the room: "this assertion is worth the entire price of admission to this course."

Step 4. Run coverage and close the gaps.

Run pytest with coverage and read the report strategically:

```
pytest --cov=payment_processor --cov-report=term-missing --cov-branch
```

Then feed the report back to AI with a prompt that targets risk, not raw numbers:

```
Here is my coverage report. Identify the three uncovered branches most
likely to contain a bug based on code complexity. For each, explain the
risk and generate a pytest test that closes it.
```

What you should get

Three additional tests, each one targeting a specific risky branch — typically the error-handling paths and the early-return guards. Re-run pytest. Coverage should now be in the 85–95% range with branch coverage above 80%. Show the green output. The room understands instantly why this is a different kind of engineering than what they did last week.

Segment 4 — Ship the Pipeline

What to Learn

A green pipeline running on every commit is the difference between a project that decays and one that improves. AI writes GitHub Actions YAML extremely well when you give it role, language, jobs, and constraints. We will generate a production-grade pipeline in one prompt, push it, watch it run, intentionally break it, and use AI to debug from the error log in under two minutes. That last move — the live debug — is the one the room will remember on Monday morning.

Step 1. Generate the pipeline.

Create the workflow directory, then ask AI for the full YAML. Notice the prompt is specific about every job and constraint:

```
mkdir -p .github/workflows
```

```
You are a senior DevOps engineer.

Generate a GitHub Actions workflow YAML for a Python 3.11 project that:
  - Triggers on push to any branch and on pull request to main.
  - Runs three jobs in PARALLEL:
      lint   – flake8 on changed files only, fail on any error
      test   – pytest with coverage, fail if branch coverage < 80%
      security – CodeQL scan, fail on any high or critical finding
  - Uses ubuntu-latest as the runner.
  - Caches pip dependencies between runs.
  - Publishes test results using the dorny/test-reporter action.
  - Prints a pass/fail summary in the job output.

Output the full YAML, ready to save as .github/workflows/ci.yml.
```

What you should get

A complete, valid ci.yml — usually 60–80 lines. Save it, commit, and push. Open the GitHub Actions tab in your browser and watch all three jobs spin up in parallel. The first run is the "magic moment" of this segment — three checks running on infrastructure you did not configure, against a codebase you refactored ten minutes ago. Let the room sit with it for ten seconds.

Step 2. Add the production-grade extras.

Once the basic pipeline is green, ask AI to harden it. This is what separates a toy pipeline from one a real team would ship:

```
Add the following to the existing workflow:
```

```
  - A matrix build that runs the test job against Python 3.11 and 3.12.
  - A separate deploy-staging job that runs only on pushes to main,
    requires all three checks to pass, and uses a GitHub Environment
    named "staging".
  - A deploy-production job that requires manual approval via a GitHub
    Environment named "production" with required reviewers configured.
  - A rollback job triggered by workflow_dispatch that redeploys the
    last successfully deployed artifact.
```

What you should get

A pipeline that now matches what real teams run in production: parallel checks, matrix coverage, environment gates, manual approval before anything reaches users, and a one-click rollback. Push the change. Open the Environments settings in GitHub and configure "production" to require your own approval. Trigger a deploy. Watch it pause. Click approve. That pause is the most important UI element in this entire module — it is the answer to "what if AI ships something bad."

Step 3. The live debug.

Pipelines break. The skill is recovering fast. Intentionally introduce a typo in the YAML — change run: pytest to run: pytset — commit, push. The workflow turns red within seconds. Now the demo move:

Copy the entire failed step's log output. Paste it into your AI chat with this prompt:

```
You are a DevOps engineer. Analyze this GitHub Actions error log and identify:
  - The root cause
  - The specific line in the YAML or configuration causing the failure
  - The exact fix to apply

<paste full error log here>
```

What you should get

Within seconds, AI returns: "command not found: pytset — typo on line 34 of ci.yml; replace pytset with pytest." Apply the fix, commit, push. Pipeline goes green again. Total time from broken to fixed: under two minutes. This is the workflow the room will reach for the next time their pipeline breaks at midnight.

Step 4. Document the pipeline.

Pipelines are infrastructure and infrastructure deserves docs. One last prompt:

```
Generate a README section for this pipeline that documents:
  - The full job topology and trigger conditions
  - All required secrets and environment variables
  - How to run each check locally
  - A runbook for the three most likely failure scenarios and their fixes
```

What you should get

A roughly 60-line README section. Paste it into the project README and commit. The next engineer who joins the team can be productive on day one instead of day five. That delta — five days saved per onboarding — is the kind of compounding return that makes the AI-native engineer worth what they get paid.

Wrap-Up — The Pipeline Is Live

Remember

In 25 minutes you took a ~100-line legacy God Class with a hidden production bug, generated a complete diagnostic, extracted a clean PaymentProcessor service using disciplined prompt chaining, generated a comprehensive test suite with 85%+ coverage including the boundary tests no human would have written by hand, and shipped a production-grade GitHub Actions pipeline with environment gates, matrix builds, and a live debug demo. That is not a toy. That is the actual workflow your students will use on real codebases starting Monday morning.

Key takeaways:

- **Diagnose before you operate.** Health report + complexity score + priority matrix takes five minutes and saves you from refactoring the wrong thing.
- **Refactor, never rewrite.** Strangler Fig + prompt chaining keeps every change small enough to review and reverse.
- **Behavioral equivalence is the safety net.** If the same tests pass before and after, you have earned the merge.
- **AI finds boundaries humans miss.** Boundary value analysis is where AI consistently outperforms manual testing — every time.
- **A green pipeline is one prompt away.** And debugging a red one is two minutes away when you paste the log into AI with the right framing.

Student Lab — Phase 1 Capstone Submission

Take the legacy application in the course repo. Run the full Module 4 pipeline on it: complete diagnostic with health report and priority matrix, refactor the top two priority issues using Strangler Fig and prompt chaining, achieve 80% or higher branch test coverage with AI-generated tests, and deploy a green GitHub Actions pipeline with lint, test, security scan, and a manual-approval production gate to your own GitHub repository.

Submit five things: the GitHub repo link, the markdown health report, the refactor pull request with inline rationale comments, a coverage screenshot showing 80%+ branch coverage, and a 3–5 minute video walkthrough explaining one architectural decision you made and why. This is the gate to Phase 2.

Appendix — order_manager.py (the legacy file used in this demo)

This is the deliberately messy ~100-line Python file that drives the entire 25-minute demo. Save it as order_manager.py at the root of the demo repo before class begins. The smells are intentional. The state mutation bug on the credit_card branch is the one the room will applaud when AI surfaces it in Segment 1.

```
# order_manager.py
# Deliberately messy. Used as the input to the Module 4 capstone demo.

class OrderManager:
    def __init__(self, db, email_client, payment_gateway, logger):
        self.db = db
        self.email_client = email_client
        self.payment_gateway = payment_gateway
        self.logger = logger
        self.report_rows = []

    def process_order(self, order_data):
        # SMELL: Long method, God class, primitive obsession (raw dicts),
        # state mutation before failure, duplicated logic between branches.
        if not order_data:
            return None
        if order_data.get("amount", 0) <= 0:
            self.logger.log("Bad amount: " + str(order_data.get("amount")))
            return None
        if not order_data.get("customer_email"):
            self.logger.log("Missing email")
            return None

        if order_data.get("payment_method") == "credit_card":
            result = self.payment_gateway.charge(
                order_data["amount"],
                order_data.get("card", {}),
            )
            if result.get("status") == "success":
                # BUG: payment is already charged at this point.
                # If inventory is short, we return None without refunding.
                for item in order_data.get("items", []):
                    inv = self.db.get_inventory(item["sku"])
                    if inv["quantity"] >= item["qty"]:
                        self.db.update_inventory(
                            item["sku"],
                            inv["quantity"] - item["qty"],
                        )
```

```
                else:
                    self.logger.log("Out of stock: " + item["sku"])
                    return None  # customer has been charged. oops.
            self.email_client.send(
                order_data["customer_email"],
                "Order confirmed",
                "Thanks for your order!",
            )
            self.report_rows.append({
                "order_id": order_data.get("id"),
                "amount": order_data["amount"],
                "method": "credit_card",
            })
            self.db.write_report(self.report_rows[-1])
            return {"status": "ok", "order_id": order_data.get("id")}
        else:
            self.logger.log("Card declined")
            return {"status": "failed", "reason": "declined"}

    elif order_data.get("payment_method") == "paypal":
        # DUPLICATED: nearly identical to the credit_card branch.
        result = self.payment_gateway.charge_paypal(
            order_data["amount"],
            order_data.get("paypal_token"),
        )
        if result.get("status") == "success":
            for item in order_data.get("items", []):
                inv = self.db.get_inventory(item["sku"])
                if inv["quantity"] >= item["qty"]:
                    self.db.update_inventory(
                        item["sku"],
                        inv["quantity"] - item["qty"],
                    )
                else:
                    self.logger.log("Out of stock: " + item["sku"])
                    return None  # same bug, different branch.
            self.email_client.send(
                order_data["customer_email"],
                "Order confirmed",
                "Thanks for your order!",
            )
            self.report_rows.append({
                "order_id": order_data.get("id"),
                "amount": order_data["amount"],
                "method": "paypal",
            })
```

```
                self.db.write_report(self.report_rows[-1])
                return {"status": "ok", "order_id": order_data.get("id")}
            else:
                self.logger.log("PayPal declined")
                return {"status": "failed", "reason": "declined"}
        else:
            self.logger.log("Unknown payment method")
            return None

    def generate_daily_report(self):
        # SMELL: this does not belong on OrderManager.
        total = 0
        for row in self.report_rows:
            total += row["amount"]
        return {"total": total, "count": len(self.report_rows)}
```