v

Module 3: Accelerated Code Generation and Debugging,
Lesson 3: Precision Debugging with AI
Hands-On: Five Broken Programs

Overview

Reading the trace before you prompt is what separates a vague question from a precise one — and precise questions get useful answers. In this Hands-On you will debug five broken programs using the five-element prompt as your accelerator. The AI is a reasoning partner, not an oracle — verify every fix before you ship it. Document your pattern cards so every expensive debugging session becomes a reusable asset.

Target: all five bugs resolved in under 60 minutes — 12 minutes per bug on average.

Before You Start

- Open your AI tool: Claude.ai, Cursor, or GitHub Copilot Chat
- Have a Python 3.11+ environment with `pip install flask sqlalchemy aiohttp` ready
- Keep the five-element prompt template visible throughout
- You may keep one AI session open per bug — or use the same session across all five

---

During the Hands-On – Keep This Visible

Five-Element Prompt Template

```
CONTEXT  —  what the code does, language, framework, version
ERROR    —  exact exception + line number, or symptom if no exception
HYPOTHESIS  —  your best current theory about the root cause
ATTEMPTS —  what you have already tried
ASK      —  the specific answer or fix you need
```

Bug 1 — Runtime TypeError in Async Code

Broken Code

```
# pricing.py
```

```
import asyncio

async def fetch_all_prices(product_ids: list[int]) -> float:
    tasks = [fetch_price(pid) for pid in product_ids]
    results = await asyncio.gather(*tasks)
    # BUG: results may contain None for unknown product IDs
    total = sum(r["price"] for r in results)
    return total

async def fetch_price(product_id: int):
    catalog = {1: {"price": 9.99}, 2: {"price": 14.99}, 3: {"price": 4.99}}
    return catalog.get(product_id)   # Returns None for unknown IDs

asyncio.run(fetch_all_prices([1, 2, 99, 3]))
```

Error Output / Stack Trace

```
Traceback (most recent call last):
  File "pricing.py", line 8, in fetch_all_prices
    total = sum(r["price"] for r in results)
  File "pricing.py", line 8, in <genexpr>
    total = sum(r["price"] for r in results)
TypeError: 'NoneType' object is not subscriptable
```

Five-Element Prompt — Paste into your AI tool

```
CONTEXT: Python 3.11 async pricing service. fetch_all_prices() fans out
price lookups concurrently using asyncio.gather() and sums the results.

ERROR: TypeError: 'NoneType' object is not subscriptable at line 8.
    total = sum(r["price"] for r in results)
Only occurs when product IDs not present in the catalog are passed in.

HYPOTHESIS: asyncio.gather() faithfully returns None for coroutines that
return None. The generator then tries to subscript None as a dict.

ATTEMPTS: Confirmed fetch_price() returns None via dict.get() for unknown
IDs. No error handling exists in the gather/sum pipeline.

ASK: What is the idiomatic fix — filter Nones before summing, use a
sentinel default in fetch_price(), or use return_exceptions=True? Show
the corrected fetch_all_prices() with production-grade None handling.
```

AI Analysis — What You Should Get

- Root cause: dict.get() returns None for missing keys; asyncio.gather() returns all coroutine results faithfully, including None. The generator then tries None["price"] and raises TypeError.
- asyncio.gather() does NOT skip or filter None-returning coroutines — None is a valid return value.
- Fix 1 (filter): sum(r["price"] for r in results if r is not None) — loses visibility on missing products.
- Fix 2 (default): return catalog.get(product_id, {"price": 0.0}) — silent; use only if 0 is a safe default.

- Fix 3 (production): filter + log warnings for None results so missing products surface in observability.
- Bonus: return_exceptions=True on gather() catches coroutine exceptions but does not change None returns.

Corrected Code

```
# CORRECTED fetch_all_prices()
async def fetch_all_prices(product_ids: list[int]) -> float:
    tasks = [fetch_price(pid) for pid in product_ids]
    results = await asyncio.gather(*tasks)
    missing = [pid for pid, r in zip(product_ids, results) if r is None]
    if missing:
        print(f"WARNING: No price found for product IDs: {missing}")
    return sum(r["price"] for r in results if r is not None)
```

Pattern Card ▸ Async None Propagation — asyncio.gather() returns all coroutine results including None; always filter or default before aggregating collected results.

Bug 2 — Logic Error — Wrong Output, No Exception

Broken Code

```
# finance.py
def calculate_compound_interest(
    principal: float,
    annual_rate: float,   # e.g. pass 5 for 5%
    years: int,
    compounds_per_year: int = 12
) -> float:
    """Returns final balance after compound interest."""
    # BUG: annual_rate not converted from percentage to decimal
    rate = annual_rate / compounds_per_year
    n    = compounds_per_year * years
    return principal * (1 + rate) ** n

result = calculate_compound_interest(10_000, 5, 10)
print(f"Final balance: ${result:,.2f}")
# Expected: ~$16,470.09
# Actual:   ~$858,388,128.56  (catastrophically wrong)
```

Error Output / Stack Trace

```
# No exception — the math runs silently with the wrong answer.

$ python finance.py
Final balance: $858,388,128.56

# Reference: correct answer for $10,000 at 5% for 10 years
# compounded monthly = $16,470.09
```

Five-Element Prompt — Paste into your AI tool

```
CONTEXT: Python 3.11 finance module. Formula: A = P(1 + r/n)^(nt)
where r is the annual rate as a decimal (5% = 0.05), n = compounds_per_year.
The caller passes annual_rate=5 meaning 5 percent.

ERROR: No exception. Wrong output.
    calculate_compound_interest(10000, 5, 10) returns ~$858,388,128.
    Expected: ~$16,470.09.

HYPOTHESIS: annual_rate is not being converted from percentage to decimal.
The function divides 5 by 12 giving rate=0.4167 instead of 0.004167,
inflating every compounding period by a factor of 100.

ATTEMPTS: Stepped through with a debugger. rate evaluates to 0.4167
at runtime, confirming the missing /100 conversion.

ASK: Confirm the hypothesis, show the corrected line, and provide a
pytest unit test with a known reference value that would catch this.
```

AI Analysis — What You Should Get

- Root cause: rate = annual_rate / compounds_per_year uses the raw percentage (5) instead of the decimal (0.05), overstating rate by exactly 100×.
- This is a unit scaling bug — the math is syntactically valid but semantically wrong, so no exception fires.
- Fix: rate = (annual_rate / 100) / compounds_per_year
- Unit tests with known reference values are the standard defence for silent numeric bugs.
- Reference value: $10,000 at 5% for 1 year compounded monthly = $10,511.62 (verifiable with any financial calculator).

Corrected Code

```
# CORRECTED calculate_compound_interest()
def calculate_compound_interest(
    principal: float,
    annual_rate: float,
    years: int,
    compounds_per_year: int = 12
) -> float:
    rate = (annual_rate / 100) / compounds_per_year  # FIXED: /100 added
    n    = compounds_per_year * years
    return principal * (1 + rate) ** n

# Regression test
def test_compound_interest_known_value():
    result = calculate_compound_interest(10_000, 5, 10)
    assert abs(result - 16_470.09) < 0.10, f"Got {result:.2f}"
```

Pattern Card ▸ Unit Scaling Bug — financial and scientific formulas can compute silently with catastrophically wrong answers; always validate against a known reference value in a unit test before shipping.

Bug 3 — ORM Query Failing — SQLAlchemy DetachedInstanceError

Broken Code

```
# models.py  (SQLAlchemy 2.0, Python 3.11)
from sqlalchemy import create_engine, Column, Integer, String, ForeignKey
from sqlalchemy.orm import declarative_base, relationship, Session

Base = declarative_base()

class User(Base):
    __tablename__ = "users"
    id     = Column(Integer, primary_key=True)
    name   = Column(String(50))
    orders = relationship("Order", lazy="select")  # default lazy load

class Order(Base):
    __tablename__ = "orders"
    id      = Column(Integer, primary_key=True)
    user_id = Column(Integer, ForeignKey("users.id"))
    total   = Column(Integer)

engine = create_engine("sqlite:///app.db")

def get_user(user_id: int) -> User:
    with Session(engine) as session:
        return session.query(User).filter(User.id == user_id).first()
        # BUG: session closes on exit; ORM object is now detached

user = get_user(1)
print(user.orders)   # Triggers lazy load on a closed session → crash
```

Error Output / Stack Trace

```
sqlalchemy.orm.exc.DetachedInstanceError:
  Instance <User at 0x7f3a...> is not bound to a Session;
  attribute refresh operation cannot proceed
  (Background on this error: https://sqlalche.me/e/14/bhk3)
```

Five-Element Prompt — Paste into your AI tool

```
CONTEXT: Python 3.11, SQLAlchemy 2.0 ORM. get_user() opens a Session
with a context manager, queries a User by ID, and returns the ORM object.
User.orders is a lazy-loaded relationship (lazy="select").

ERROR: sqlalchemy.orm.exc.DetachedInstanceError on user.orders access.
  Instance <User> is not bound to a Session; attribute refresh cannot proceed.

HYPOTHESIS: The `with Session(...) as session:` block closes the session
when get_user() returns. Accessing user.orders outside that scope triggers
a lazy load that requires an active session — which no longer exists.
```

```
ATTEMPTS: Confirmed the error fires only on user.orders, not user.name
(name was already loaded into the instance __dict__ before session closed).

ASK: Show three options to fix this — (1) eager loading, (2) expunge/detach
pattern, (3) session-per-request for FastAPI — and explain the trade-offs.
```

AI Analysis — What You Should Get

- Root cause: SQLAlchemy's lazy='select' defers loading relationships until first access. Closing the session before that access detaches the instance.
- user.name already in __dict__ loads fine; user.orders has never been fetched so it needs the session.
- Fix 1 — Eager load (best for read endpoints): add joinedload(User.orders) inside the query.
- Fix 2 — selectinload: issues a second SELECT IN query; avoids a JOIN; preferred for collections.
- Fix 3 — FastAPI pattern: use Depends(get_db) to yield a session alive for the full request scope.
- Avoid returning raw ORM objects from closed sessions; return Pydantic models instead.

Corrected Code

```
# CORRECTED — Option 1: eager load with selectinload (recommended)
from sqlalchemy.orm import selectinload

def get_user(user_id: int) -> User:
    with Session(engine) as session:
        user = (
            session.query(User)
            .options(selectinload(User.orders))  # load before session closes
            .filter(User.id == user_id)
            .first()
        )
        session.expunge(user)   # safely detach with all data loaded
        return user
```

Pattern Card ▸ ORM Lazy Load Lifecycle — never return a lazy-loaded ORM object from a closed session; eager-load all required relationships inside the session scope or keep the session alive for the full request.

Bug 4 — Missing await — Event Loop Corruption

Broken Code

```
# notifications.py
import asyncio
```

```
async def send_notification(user_id: int) -> dict:
    await asyncio.sleep(0.1)   # simulates network call
    return {"user_id": user_id, "status": "sent", "channel": "email"}

async def notify_all_users(user_ids: list[int]) -> list[dict]:
    # BUG: asyncio.gather() is a coroutine — missing await
    results = asyncio.gather(*[send_notification(uid) for uid in user_ids])

    # results is a coroutine object, not a list — TypeError below
    sent = [r for r in results if r["status"] == "sent"]
    return sent

asyncio.run(notify_all_users([101, 102, 103]))
```

Error Output / Stack Trace

```
Traceback (most recent call last):
  File "notifications.py", line 12, in notify_all_users
    sent = [r for r in results if r["status"] == "sent"]
  File "notifications.py", line 12, in <listcomp>
TypeError: 'coroutine' object is not subscriptable

sys:1: RuntimeWarning: coroutine 'gather' was never awaited
RuntimeWarning: Enable tracemalloc to get the object allocation traceback
```

Five-Element Prompt — Paste into your AI tool

```
CONTEXT: Python 3.11 async notification service. notify_all_users()
fans out send_notification() calls concurrently with asyncio.gather(),
then filters results to confirm successful sends.

ERROR: TypeError: 'coroutine' object is not subscriptable at line 12.
RuntimeWarning: coroutine 'gather' was never awaited.

HYPOTHESIS: I forgot to await asyncio.gather(). Without await, gather()
returns a coroutine object. The list comprehension iterates over the
coroutine instead of the resolved list of dicts.

ATTEMPTS: Verified send_notification() works correctly when called with
await in isolation. The bug is in notify_all_users() only.

ASK: Confirm the one-line fix. Also explain what Python's async runtime
does with an unawaited coroutine — does it execute at all? Does the
RuntimeWarning always fire? Are there cases where this fails silently?
```

AI Analysis — What You Should Get

- Root cause: asyncio.gather() is itself a coroutine function. Without await, it returns a coroutine object and schedules nothing.
- None of the send_notification() coroutines ever execute — no notifications are sent, no exceptions raised inside them.
- Python emits RuntimeWarning: coroutine was never awaited on garbage collection, but this can be missed in production logs.
- Fix: results = await asyncio.gather(*[send_notification(uid) for uid in user_ids])

- This is a silent failure class in production: no side effects, no crash until the caller tries to use the results.
- Linters (pylint, mypy) and type checkers will flag unawaited coroutines if async stubs are correct.

Corrected Code

```
# CORRECTED notify_all_users()
async def notify_all_users(user_ids: list[int]) -> list[dict]:
    results = await asyncio.gather(   # FIXED: await added
        *[send_notification(uid) for uid in user_ids],
        return_exceptions=True         # prevents one failure killing all
    )
    sent = [
        r for r in results
        if isinstance(r, dict) and r.get("status") == "sent"
    ]
    return sent
```

Pattern Card ▸ Unawaited Coroutine Sink — asyncio.gather() without await silently drops all work and returns a coroutine object; treat RuntimeWarning: never awaited as a P0 production alert.

Bug 5 — Multi-Service Interaction — 500 KeyError

Broken Code

```
# orders.py  (Flask 3.0)
from flask import Flask, request, jsonify

app = Flask(__name__)

@app.route("/api/orders/process", methods=["POST"])
def process_order():
    payload    = request.get_json()
    line_items = payload["line_items"]

    for item in line_items:
        # BUG: we read "available_qty" but inventory service sends "stock_count"
        if item["available_qty"] < item["quantity"]:
            return jsonify({"error": f"Insufficient stock for {item['sku']}"}), 400

    total = sum(item["unit_price"] * item["quantity"] for item in line_items)
    return jsonify({"status": "confirmed", "total": total}), 200

# Upstream inventory service actually sends:
# {"line_items": [
#   {"sku": "WIDGET-A", "quantity": 2, "unit_price": 9.99, "stock_count": 10}
# ]}
```

Error Output / Stack Trace

```
[2024-01-15 14:23:07] ERROR in app: KeyError: 'available_qty'
Traceback (most recent call last):
  File "/app/routes/orders.py", line 12, in process_order
    if item['available_qty'] < item['quantity']:
KeyError: 'available_qty'
127.0.0.1 - - [15/Jan/2024 14:23:07]
  "POST /api/orders/process HTTP/1.1" 500 -
```

Five-Element Prompt — Paste into your AI tool

```
CONTEXT: Flask 3.0 order processing service. Receives a POST from an
upstream inventory service, validates stock levels, then confirms the order.
Two teams own the services and deploy independently.

ERROR: 500 Internal Server Error. Flask logs show KeyError: 'available_qty'
at line 12. The upstream service is sending well-formed JSON — the error
is in how our code reads the payload.

HYPOTHESIS: Contract mismatch. Our code reads item["available_qty"] but
logging the raw request.get_json() payload shows the upstream sends
"stock_count". This is a field name disagreement between two teams.

ATTEMPTS: Logged raw payload — field is confirmed as "stock_count" in
all real requests. Upstream team will not rename the field.

ASK: (1) Show the immediate one-line fix. (2) What is the correct
long-term approach to prevent cross-service key mismatches? (3) How
should this endpoint handle bad payload shapes as a 422, not a 500?
```

AI Analysis — What You Should Get

- Root cause: item['available_qty'] raises KeyError because the upstream service sends 'stock_count'. This is a contract drift bug — two teams diverged on the field name.
- Immediate fix: change item['available_qty'] to item['stock_count'] (or .get('stock_count', 0) for defensive access).
- A raw dict KeyError should never escape as an unhandled 500; all external payload access must go through validation.
- Long-term fix: validate the incoming payload with a Pydantic model at the endpoint boundary — gives a clean 422 on schema violations.
- Contract management: adopt OpenAPI as the shared contract between teams; enforce field names in CI with contract testing (e.g., Pact, Schemathesis).
- Defensive pattern: item.get('stock_count', 0) silences the crash but hides the mismatch — prefer Pydantic validation that fails loudly.

Corrected Code

```
# CORRECTED — Pydantic validation at the boundary
from pydantic import BaseModel
from typing import List

class LineItem(BaseModel):
    sku:         str
```

```
    quantity:    int
    unit_price:  float
    stock_count: int   # matches upstream field name exactly

class OrderPayload(BaseModel):
    line_items: List[LineItem]

@app.route("/api/orders/process", methods=["POST"])
def process_order():
    try:
        payload = OrderPayload.model_validate(request.get_json())
    except Exception as e:
        return jsonify({"error": "Invalid payload", "detail": str(e)}), 422

    for item in payload.line_items:
        if item.stock_count < item.quantity:  # now a typed attribute
            return jsonify({"error": f"Insufficient stock for {item.sku}"}), 400

    total = sum(i.unit_price * i.quantity for i in payload.line_items)
    return jsonify({"status": "confirmed", "total": round(total, 2)}), 200
```

Pattern Card ▸ Cross-Service Contract Drift — never use raw dict access on payloads from external services; validate at the boundary with Pydantic so field mismatches surface as 422s, not 500s.

Wrap-Up — Your Pattern Card Collection

Each pattern card below is a reusable asset. After this lab you should have five cards ready to carry into every future debugging session.

Bug 1 — Async None Propagation
asyncio.gather() returns all coroutine results including None; always filter or default before aggregating.

Bug 2 — Unit Scaling Bug
Financial and scientific formulas can silently compute catastrophically wrong answers; validate against a known reference value in a unit test.

Bug 3 — ORM Lazy Load Lifecycle
Never return a lazy-loaded ORM object from a closed session; eager-load required relationships inside the session scope.

Bug 4 — Unawaited Coroutine Sink
asyncio.gather() without await silently drops all work; treat RuntimeWarning: never awaited as a P0 production alert.

Bug 5 — Cross-Service Contract Drift
Never use raw dict access on external payloads; validate at the boundary with Pydantic so mismatches surface as 422s, not 500s.

Remember — reading the trace before you prompt is what separates a vague question from a precise one. Async and distributed errors require more context than single-function bugs, so give them more. Verify every fix before you ship it. The AI is a reasoning partner, not an oracle.