

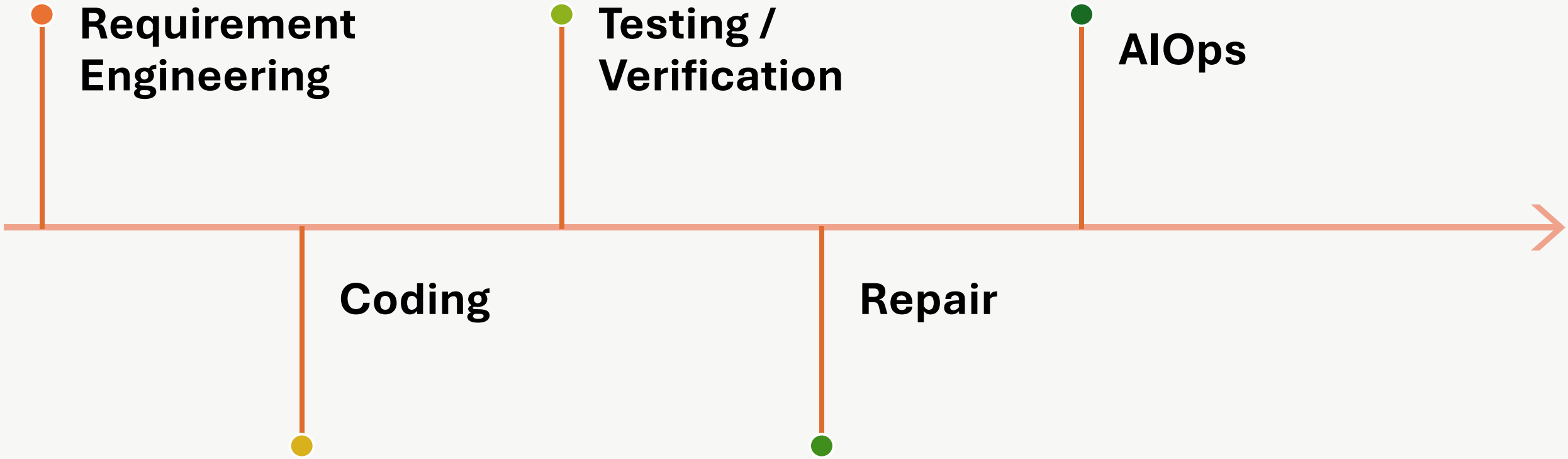
香港中文大學（深圳）
The Chinese University of Hong Kong, Shenzhen

CSC4801
AI-assisted Software Engineering

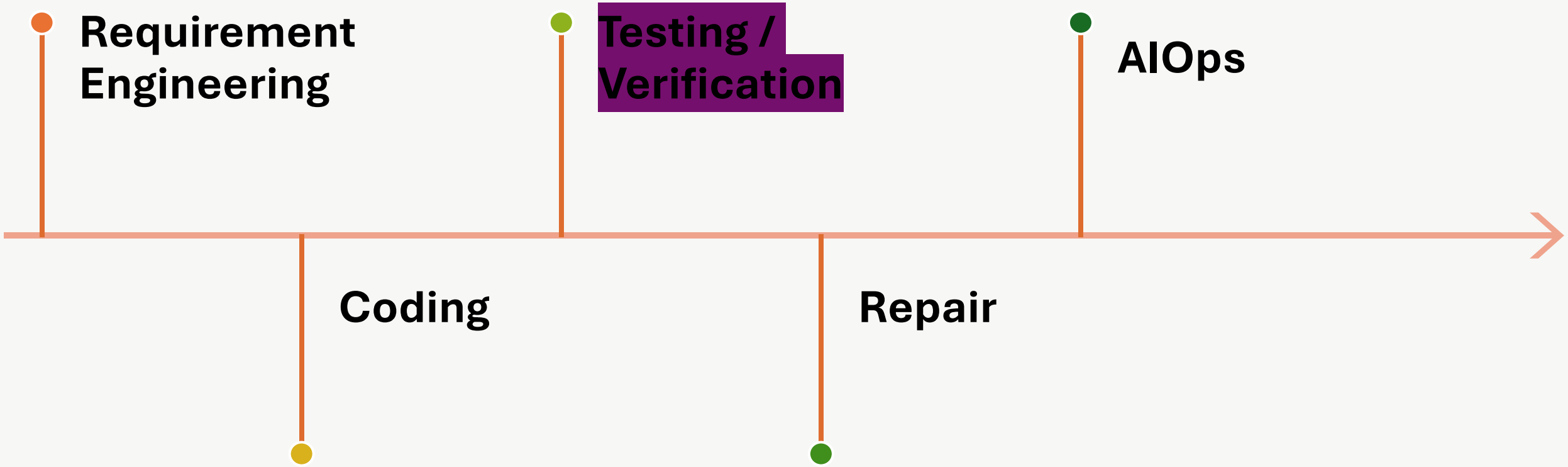
Lecture 14: Validation Exercise

Instructor: Jinsheng Ba | 2026 Fall

AI-Assisted Software Engineering



AI-Assisted Software Engineering



Goal

- Exercise 1: AI-Assisted Test Generation
- Use an LLM to generate test cases for a given function
- Compare LLM-generated tests vs. manual tests
-
- Exercise 2: Write a Fuzzer
- Find hidden bugs using AI-assisted fuzzing
- Practice: generate inputs that trigger edge cases
-
- Exercise 3: Assertion Writing
- Use LLMs to write assertions for existing code
- Evaluate assertion quality and coverage

AI-Assisted Test Generation

Test Case Generation

- Choose a function with non-trivial logic:
  - A sorting function with edge cases
  - A parser that handles multiple formats
  - A validation function with complex rules

  Make sure you know the expected behavior.

Workflow

1 Specify	Write the contract and calculate a few expected results yourself.
2 Generate	Ask the LLM for missing partitions and runnable candidate tests.
3 Execute	Review the tests, run them, and retain the actual failure.
4 Verify	Apply a narrow fix; rerun regressions and independent checks.

First decide what "correct" means

Contract	Inputs, outputs, exceptions, side effects, and boundaries.
Oracle	A hand calculation, trusted reference, or justified property.
Ambiguity	Ask for clarification; record assumptions before generating tests.
Example	"Sort correctly" is incomplete: must the input remain unchanged?

Lab 1 | Shipping: the contract

- Inputs: exact int types; bool, float, and str raise TypeError.
- Values: subtotal >= 0; weight is 1..20 kg. Otherwise ValueError.
- At subtotal >= 5000 cents: fee = 0, after validating inputs.
- Otherwise: fee = 500 + 100 * (weight - 1) cents.

Read the implementation with suspicion

```
def shipping_fee(subtotal, weight):
    if not isinstance(subtotal, int) or not
isinstance(weight, int):
        raise TypeError("integers required")
    if subtotal < 0 or not 0 <= weight <= 20:
        raise ValueError("out of range")
    if subtotal > 5000:
        return 0
    return 500 + 100 * (weight - 1)
```

Predict before running:

(5000, 1)

(100, 0)

(True, 1)

Which requirement justifies each prediction?

Manual first: build a small oracle table

(100, 20) -> 2400	Maximum valid weight; calculate the surcharge.
(5000, 1) -> 0	Exactly at the free-shipping threshold.
(100, 0) -> ValueError	Just below the minimum valid weight.
(True, 1) -> TypeError	Python bool is a subclass of int; the contract rejects it.

Prompt for partitions, not just "10 tests"

```
Generate Python unittest tests using only the standard library.
First map each requirement to valid, boundary, and invalid cases.
Calculate expected fees from the contract; show the arithmetic.
Check exact exception types, including bool rejection.
Do not infer expected outputs from the implementation.
List ambiguities; do not change production code.
```

Review the generated tests before you run

```
import unittest
from shipping import shipping_fee

class ShippingTests(unittest.TestCase):
    def test_threshold(self):
        self.assertEqual(shipping_fee(5000, 1), 0)

    def test_upper_weight(self):
        self.assertEqual(shipping_fee(100, 20),
2400)
```

Check:

Imports and API exist.

Expected values match the contract.

Assertions can fail.

Exceptions are specific.

Cases add distinct value.

Write a Fuzzer

Lab 2 | Fuzz a range parser

- Input: comma-separated integers or inclusive a-b intervals.
- Endpoints: 0..1000, with a <= b; whitespace around tokens is OK.
- Output: sorted, distinct integers. Example: "1-3,2,5" -> [1,2,3,5].
- Malformed input -> ValueError; non-string -> TypeError.

A parser can return the wrong answer without crashing

```
bounds = [int(v) for v in token.split("-")]
lo, hi = bounds[0], bounds[-1]
if not 0 <= lo <= hi <= 1000:
    raise ValueError("out of range")
if len(bounds) == 1:
    result.add(lo)
else:
    result.update(range(lo, hi))

# Full input validation is in ranges.py.
```

Predict:

"3" -> ?

"1-3" -> ?

"0-0" -> ?

A useful fuzzer needs a failure oracle.

Prompt for a bounded, reproducible fuzzer

```
Write a standard-library Python fuzzer for parse_ranges.
Generate 1000 valid single intervals in the domain 0..1000.
Use random.Random(4801); keep runs bounded.
Compare output with the inclusive interval specified by the contract.
Report seed, trial, failing input, and expected/actual behavior.
Do not swallow unexpected exceptions.
Then propose a smaller input and recheck the failure.
Explain which parts of the parser contract remain untested.
```

Demonstration | Structured inputs plus an oracle

```
rng = random.Random(4801)
for trial in range(1000):
    lo = rng.randint(0, 999)
    hi = rng.randint(lo, 1000)
    text = f"{lo}-{hi}"
    got = parse_ranges(text)
    expected = list(range(lo, hi + 1))
    if got != expected:
        print(trial, text, len(expected), len(got))
        break
```

Why this works:

Mostly meaningful inputs.

An exact semantic check.

A fixed seed.

A bounded run.

Run: python fuzz_ranges.py

Write a Real-world Fuzzer

The LibAFL Fuzzing Library

by Andrea Fioraldi and Dominik Maier



Welcome to LibAFL, the Advanced Fuzzing Library. This book shall be a gentle introduction to the library.

This version of the LibAFL book is coupled with the release 1.0 beta of the library.

This document is still work-in-progress and incomplete. The structure and the concepts explained here are subject to change in future revisions, as the structure of LibAFL itself will evolve.

The HTML version of this book is available online at https://aflplus.plus/libafl-book/ and offline from the LibAFL repository in the `docs/` folder. Build it using `mdbook build` in this folder, or run `mdbook serve` to view the book.

https://aflplus.plus/libafl-book/

Assertion Writing

Ask LLM to Write Assertions

- Give the function code and ask for assertions.
  Be specific about what to check:
  Return value, side effects, state changes, error handling.

```
prompt = """
def process_order(order):
    # implementation

Write assertions that verify:
1. Return value is correct type and structure
2. Side effects are as expected
3. Error cases are handled properly
4. State mutations are correct
"""
```

Evaluate Assertion Quality

- Are the assertions meaningful or trivial?
  Do they catch real bugs if you introduce them?
  Would you keep these assertions in production code?

Lab 3 | Transfer: check the whole state

- Success: debit src, credit dst, return None, preserve total and other accounts.
- Reject: non-int or bool, nonpositive amount, same or missing account.
- Reject insufficient funds with ValueError; every rejection preserves the entire dict.
- Assume valid initial balances and no concurrent updates.

An exception can still leave corrupted state

```
# After validating types, accounts, and positive
amount:
balances[src] -= amount
if balances[src] < 0:
    raise ValueError("insufficient funds")
balances[dst] += amount

# Before: {"A": 100, "B": 20}
# Call: transfer(balances, "A", "B", 101)
# Exception raised -- but what happened to balances?
```

Check more than:

"Did it raise?"

Also check:

"Did the failed operation leave the state unchanged?"

Replace weak assertions with specific obligations

Too weak	assert balances | The dict can be nonempty and still be wrong.
Success check	A=70, B=50, C=7 after transferring 30 from A=100 to B=20.
Failure check	Specific exception AND balances == before.copy().
Production code	Use explicit validation and exceptions; bare assert is not input validation.

Prompt for assertions that expose side effects

```
Given the transfer contract, generate Python unittest tests.
Use fresh balances for each case; snapshot with balances.copy().
Check exact account deltas, untouched accounts, total, and None return.
For every rejected call, assert the exact exception type AND
verify the whole dict is unchanged after the exception block.
Include insufficient funds, exact balance, and bool amount.
For each assertion, name a plausible bug it would detect.
Do not replace assertions with checks that merely repeat the code.
```

Worked test | Exception plus unchanged state

```
def test_rejected_transfer_is_atomic(self):
    balances = {"A": 100, "B": 20}
    before = balances.copy()

    with self.assertRaises(ValueError):
        transfer(balances, "A", "B", 101)

    self.assertEqual(balances, before)
```

Two obligations:

1. Correct rejection.

2. No state mutation.

The starter meets only the first.

Fallback: test_transfer.py

Do the tests detect faults? Try mutation

Threshold	Change >= 5000 to > 5000.
Domain and type	Allow zero weight; allow bool as an int.
Arithmetic	Change base fee 500 to 600; omit the weight surcharge.
Measure	Killed mutants / non-equivalent mutants tested; inspect survivors.

Hand in evidence, not a screenshot of an answer

- For each lab: requirement -> test -> failure -> fix -> rerun.
- Include the prompt, reviewed changes, and independently checked expected values.
- Report a minimal witness, command output, and fuzz seed when applicable.
- State one remaining gap; compare manual and expanded tests fairly.

Exit ticket | Would you accept this patch?

- The LLM says: "All ten generated tests pass, so the code is correct."
- The patch removes a validation check that caused a failing test.
- Name one reason to reject the claim and two checks you would run.



香港中文大學（深圳）
The Chinese University of Hong Kong, Shenzhen

Questions

Next class: Security