

Kinds of Invariants

C++ Under the Sea

October 2026 · The Netherlands

Victor Ciura

Principal Architect
Microsoft

Quiz: What does this print?

```
template <typename T>
struct interval { T low; T high; };

template <typename T>
T my_clamp(T const& value, interval<T> const& range) {
    return std::clamp(value, range.low, range.high);
}

int main() {
    interval<int> bad_range{10, 5};
    std::cout << my_clamp(7, bad_range) << '\n';
}
```

A) 5

B) 7

C) 10

D) it crashes

Guaranteed: nothing!

Depending on your std lib implementation and compiler, any of:

- Assertion '`!(__hi < __lo)`' failed. (abort)
- prints `5`
- prints `10`
- Bad bounds passed to `std::clamp` (with stdlib hardening, abort)

Same source, different outcomes. Anything possible!

Hot take: UB is rarely where the bug *is*. It's where a broken promise finally *lands*.

`clamp` has a "hidden" precondition. Our `interval` never stated `low <= high`, let alone enforced it.

Must see: "How Invariant are Invariants?"



What you'll take home

- **A vocabulary:** the *kinds* of invariants, and who relies on each of them
- **A mental model:** the envelope, the language rules, your types promises, and where UB comes from
- **A toolbox:** *state* it, *encode* it, *enforce* it, *monitor* it
- **A strategy for the agentic era:** challenge the evidence, govern the automation

Not a C++26 Contracts tutorial. Not another "UB is scary" talk :)

An invariant: a condition that must *always* hold

"Always" is doing a lot of work:

- *for whom?* the compiler, your callers
- *where?* memory, data, state, concurrency, environment
- *when?* on every operation, or only between public calls?

Class invariants (the classic):

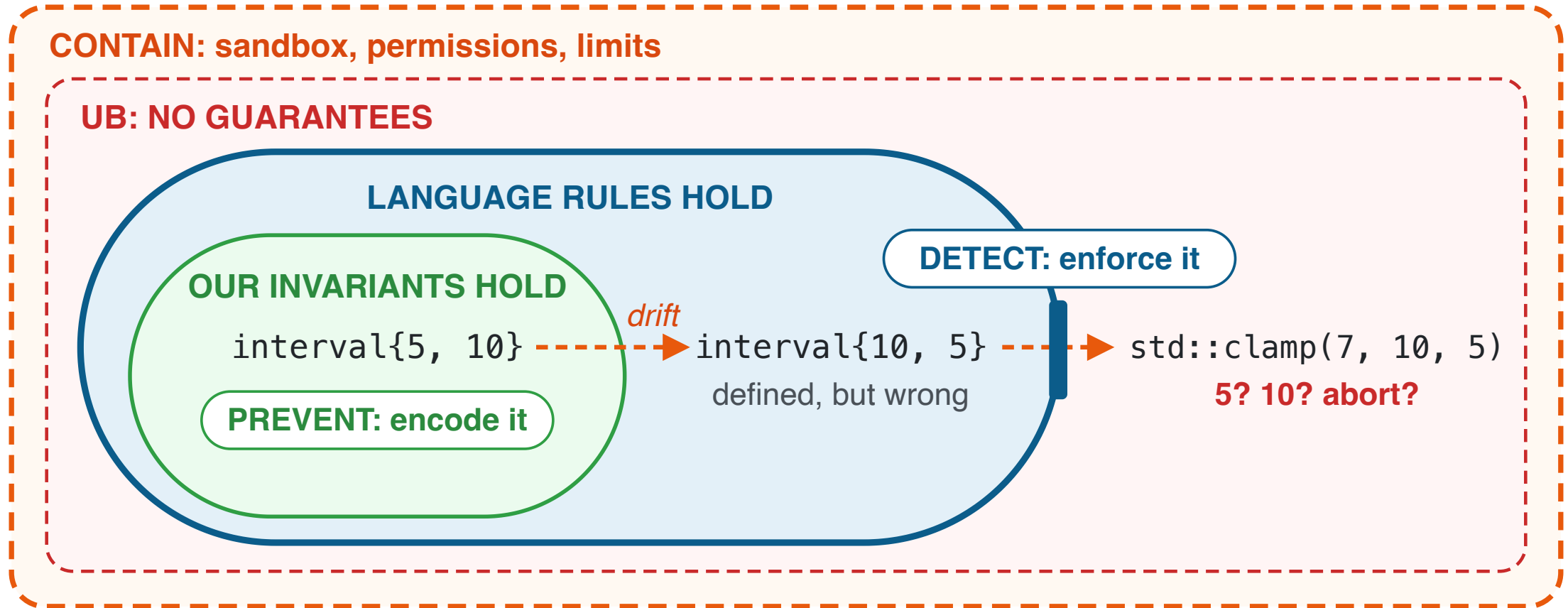
- *established* by every constructor
- *assumed* on entry to every public member
- *re-established* on exit from every public member

vector : `size() <= capacity()`

set : elements are sorted

Invariants turn **assumptions** (liabilities) into **engineering assets**: local reasoning, fearless refactoring, aggressive optimization, safe delegation.

Validity, safety, undefined behavior



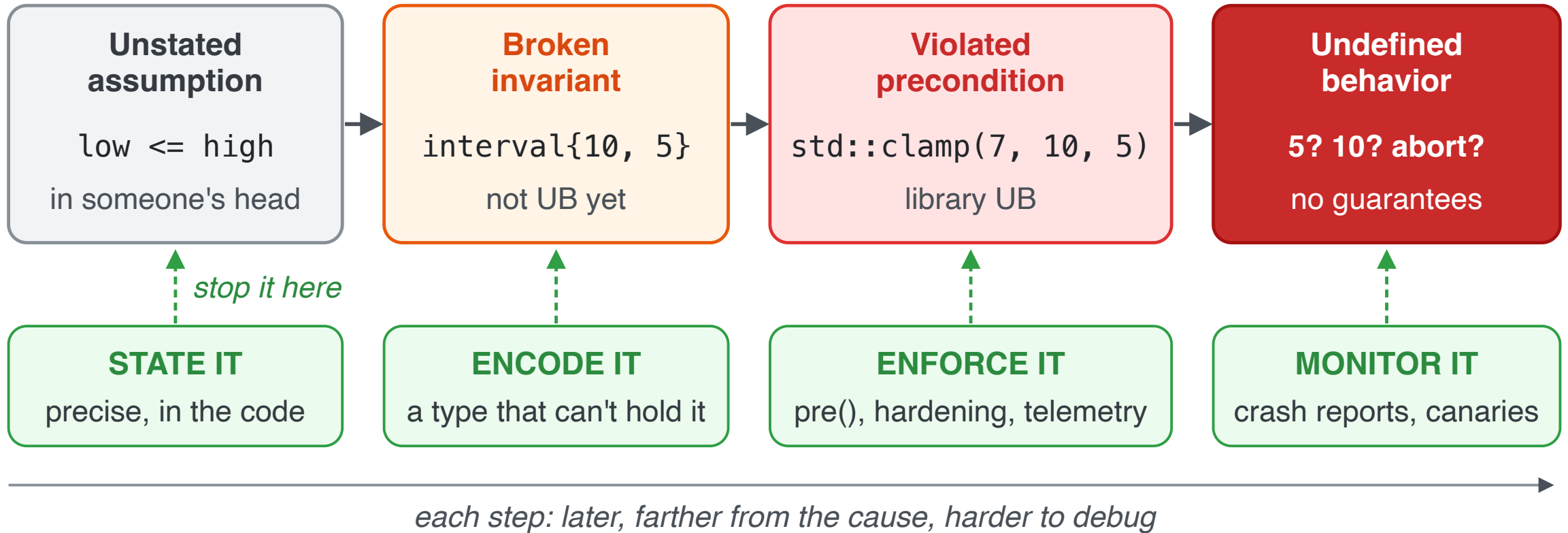
Validity: inside the intended envelope. **Safety**: failures prevented or contained when validity is threatened. **UB**: a language rule broken; no guarantees.

Two kinds of invariants: who relies on them?

	Language rules	Your type's promise
Relied on by	the compiler and the optimizer	the users of your abstraction
Defined by	the language	you
Must hold	on every operation	between public calls
If broken	UB at that operation	wrong now; can become UB later
Examples	no access outside lifetime; no data races; no signed overflow	<code>size() <= capacity(); low <= high</code>

Language rules: always. Your promises: at every boundary.

How a broken invariant becomes UB



A broken promise is a **time bomb**: it can go off *later, elsewhere*, in *someone else's* code.

Kinds of invariants, by layer

LAYER	ONE EXAMPLE	OWNER / IF BROKEN
Environment	config is sane; clocks are monotonic	the system, ops outages, drift
Concurrency	no data races; one lock for both bounds	language + your design data race: UB mixed snapshot: wrong
State	connecting -> connected; open before read	your types, protocols wrong results; UB possible
Data	low <= high; units match; valid UTF-8	your types wrong results; UB possible
Memory	in bounds; alive while used; the right type	the language UB when accessed

Thesis

Invariants are contracts, not comments

1

Specify

the promise

behavior and
obligations

2

Encode

the invariants

valid states,
transitions

3

Constrain

the system

boundaries,
capabilities

4

Challenge

the evidence

mutation tests,
adversaries

5

Govern

the automation

humans own
the intent

Make invalid states harder to represent and easier to detect

PILLAR 1

Specify the promise

Define required behavior, caller obligations, and measurable acceptance criteria.

Specify

Encode

Constrain

Challenge

Govern

Every public function is a promise

A **contract** says which inputs and state a function requires to have *defined behavior*, and what effect it promises to deliver.

	Who must uphold it	For <code>my_clamp(value, range)</code>
Precondition	the caller	<code>range.low <= range.high</code>
Postcondition	the function	the result is within <code>[range.low, range.high]</code>
Invariant	the type, between calls	<code>low() <= high()</code>

Make the promise precise enough to challenge.

Mechanize the checks you can; **document & review** the obligations you can't.

Contracts, not comments: Say it in code!

```
int my_clamp(int value, interval<int> const& range)
  pre  (range.low <= range.high)
  post (r: range.low <= r && r <= range.high)
{
  return std::clamp(value, range.low, range.high);
}
```

```
contract violation in function int my_clamp(int, const interval<int>&)
at example.cpp:8: range.low <= range.high
[assertion_kind: pre, semantic: enforce, ..., terminating: yes]
```

ignore: no check. **observe**: report, continue. **enforce**: report, terminate. **quick-enforce**: terminate.
The *implementation* picks the semantic (a build flag), not the source.

Same invariant, two different promises

Wide: every argument value is handled

```
interval(T lo, T hi)
: lo_(std::move(lo)),
  hi_(std::move(hi))
{
    if (not (lo_ <= hi_))
        throw std::invalid_argument(
            "low > high");
}
```

Bad input is an *expected event*.
Natural at **trust boundaries**.

Narrow: the caller promises

```
interval(T lo, T hi)
    pre (lo <= hi)
: lo_(std::move(lo)),
  hi_(std::move(hi))
{ }
```

Bad input is a *bug* in the caller.
Natural **inside** the boundary.

A narrow contract can be **widened later** without breaking anyone.
Narrowing a wide one is a breaking change.

PILLAR 2

Encode the invariants

Represent valid states and permitted transitions explicitly, including exceptional and partially usable states.

Specify

Encode

Constrain

Challenge

Govern

Encode it: from `struct` to `class`

```
template <typename T> requires std::totally_ordered<T> && std::movable<T>
class interval {
    T lo_, hi_;
public:
    interval(T lo, T hi) : lo_(std::move(lo)), hi_(std::move(hi)) {
        if (not (lo_ <= hi_))
            throw std::invalid_argument("interval: low > high");
    }
    T const& low() const { return lo_; }
    T const& high() const { return hi_; }
};
```

- **Private members**, one checking constructor: check once, trust everywhere after
- `my_clamp` drops its bounds-order precondition: the type carries the proof
- **Concepts**: `interval<std::complex<float>>` won't compile; `interval<double>` will (NaN!)

Question: is `low() <= high()` *always* true now?

```
using namespace std::literals;

interval<std::pair<std::string, int>> range{ {"abc"s, 99}, {"xyz"s, 42} };
interval<std::pair<std::string, int>> range2{ std::move(range) };

assert(range2.low() <= range2.high());
assert(range.low() <= range.high());
```

**Checked constructor. Private members. No setters.
Can it still break?**

The moved-from object breaks the invariant

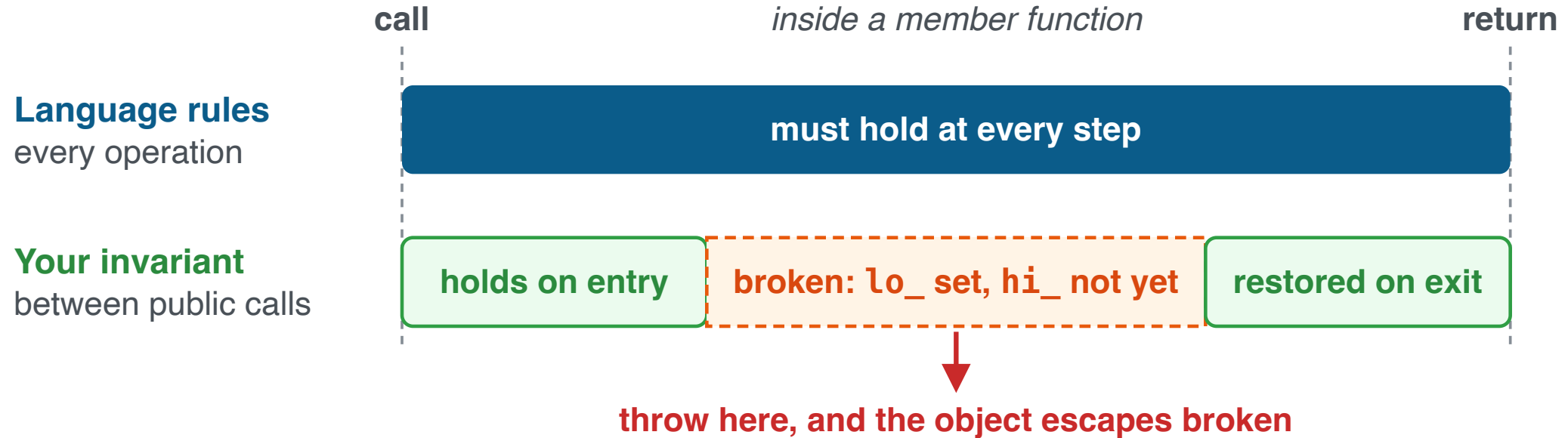
```
range2:          [ ("abc", 99), ("xyz", 42) ]  
moved-from range: [ ("", 99), ("", 42) ]  
Assertion failed: (range.low() <= range.high())
```

- The generated move constructor moves **member by member**
- Each string is left "valid but unspecified": here, both are now empty
- But `("", 99) <= ("", 42)` is **false**: the *relationship* between the members did not survive

Member-wise operations don't preserve relationships *between* members.

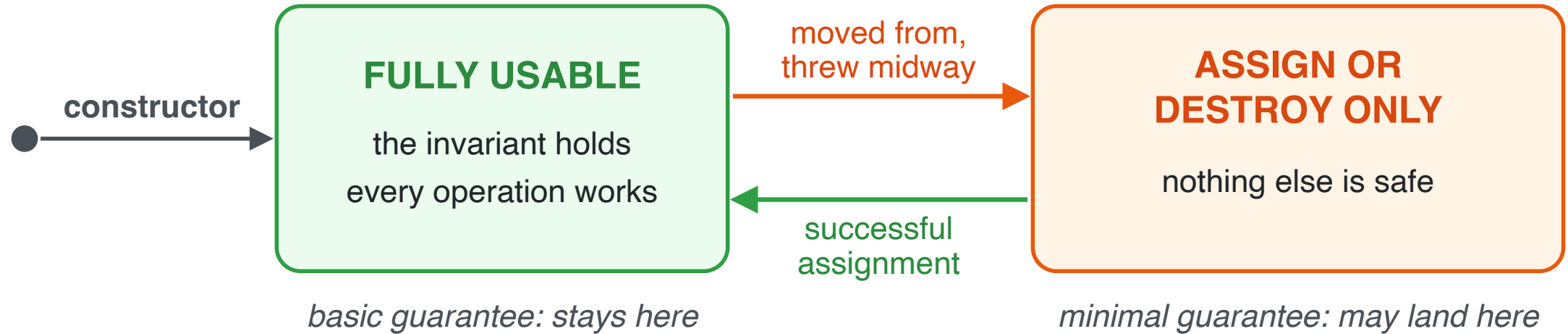
Moved-from and **exception-tainted** objects are where invariants break.

Why: invariants also break *inside* the class



```
interval& operator=(interval const& z) {  
    lo_ = z.lo_;    // the invariant may be broken from here...  
    hi_ = z.hi_;    // ...and if this copy throws, it stays broken  
    return *this;  
}
```

After a move or a throw: how usable is it?



Fully usable: the invariant holds; operations work as specified. A moved-from `std::string` is: "valid but unspecified".

Assign or destroy only: our moved-from `interval<pair<string, int>>`

Make the partial state explicit and queryable

```

class interval {
    bool has_value_; T lo_, hi_;
public:
    interval(interval&& z) noexcept(std::is_nothrow_move_constructible_v<T>)
        : has_value_(std::exchange(z.has_value_, false)),
          lo_(std::move(z.lo_)), hi_(std::move(z.hi_)) {}
    interval& operator=(interval const& z) {           // commit last
        if (this == &z) return *this;
        has_value_ = false;
        if (z.has_value_) { lo_ = z.lo_; hi_ = z.hi_; has_value_ = true; }
        return *this;
    }
    bool has_value() const noexcept { return has_value_; }
    T const& low() const pre (has_value()) { return lo_; } // high(): same
};

```

Relaxed, but honest :)

Encode states and transitions, not just values

Before: flags and hope :)

```
struct connection {
    bool connecting = false;
    bool connected  = false;
    socket sock;      // valid?
    time_point started; // set?
    int retries = 0;
};
```

Several flag combinations, some of them nonsense.

Which fields are valid? Ask the *comments*.

One dispatcher owns the `connection` and applies the *state transitions*.

After: each state carries only its own data

```
struct disconnected { int retries; };
struct connecting  { time_point started;
                    int retries; };
struct connected   { socket sock; };

using connection = std::variant<
    disconnected, connecting, connected>;

connected    on_established(connecting&&,
                           socket);
disconnected on_timeout(connecting&&);
```

Two atomics do not make one invariant

```
std::atomic<int> lo{1}, hi{5};

void writer() { lo.store(10); hi.store(20); } // [1, 5] -> [10, 20]
interval<int> reader() { return {lo.load(), hi.load()}; } // version #1
```

A legal schedule	reader() gets
both loads between the writer's two stores	[10, 5] : the constructor throws
lo before the update, hi after it	[1, 20] : passes the check, but never published

No data race, no UB: every language rule holds. **Our promise doesn't.**

Ordering is not a transaction: a snapshot must be *one* published version.

Protect the relation, not the fields

```
mutable std::mutex mu_;  
interval<int> range_{1, 5};
```

Publish one valid value

```
void replace(interval<int> next) {  
    std::lock_guard lock{mu_};  
    range_ = next;  
}
```

Copy out one snapshot

```
interval<int> snapshot() const {  
    std::lock_guard lock{mu_};  
    return range_;  
}
```

Every reader and writer uses **the same mutex**. No reference to `range_` escapes.

PILLAR 3

Constrain the system

Prevent or contain failures through safe abstractions, restricted capabilities, and resource limits.

Specify

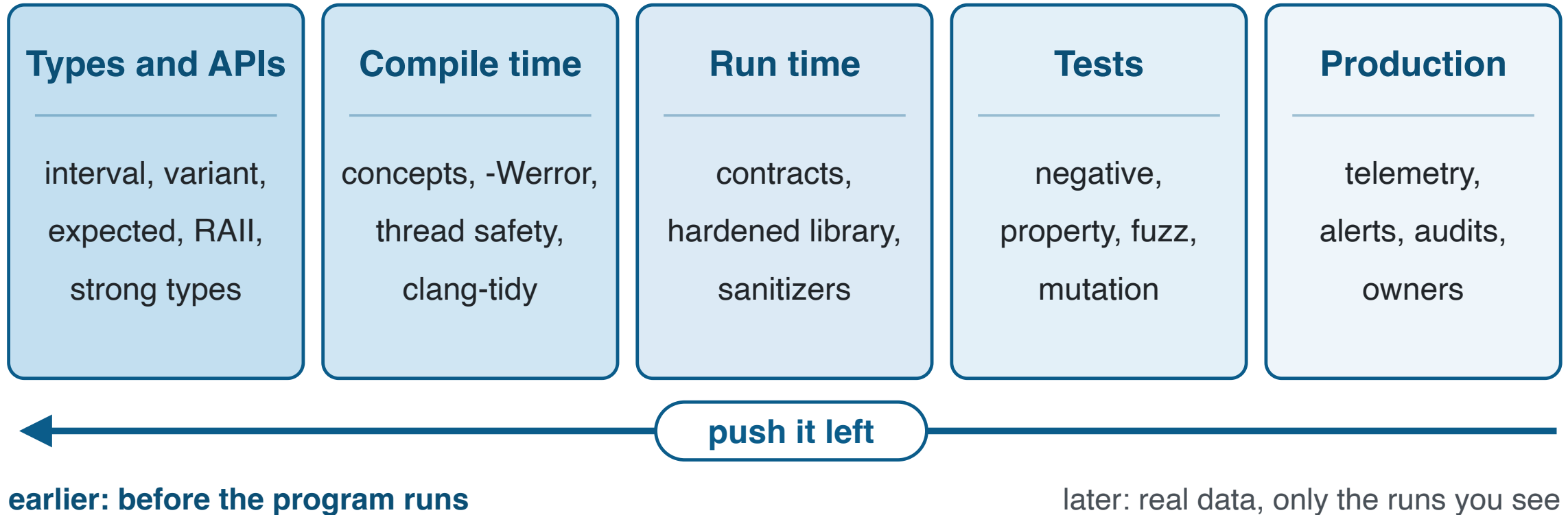
Encode

Constrain

Challenge

Govern

Push each invariant as far LEFT as it will go



Let the machine check it, every build

Concurrency: Tools - *thread-safety analysis*

```
class account {  
    mutable std::mutex mu_;  
    long balance_ GUARDED_BY(mu_) = 0;  
public:  
    void deposit(long amount) {  
        std::lock_guard lock(mu_);  
        balance_ += amount;  
    }  
    long peek() const { return balance_; }  
};
```

```
warning: reading variable 'balance_' requires  
holding mutex 'mu_' [-Wthread-safety-analysis]
```

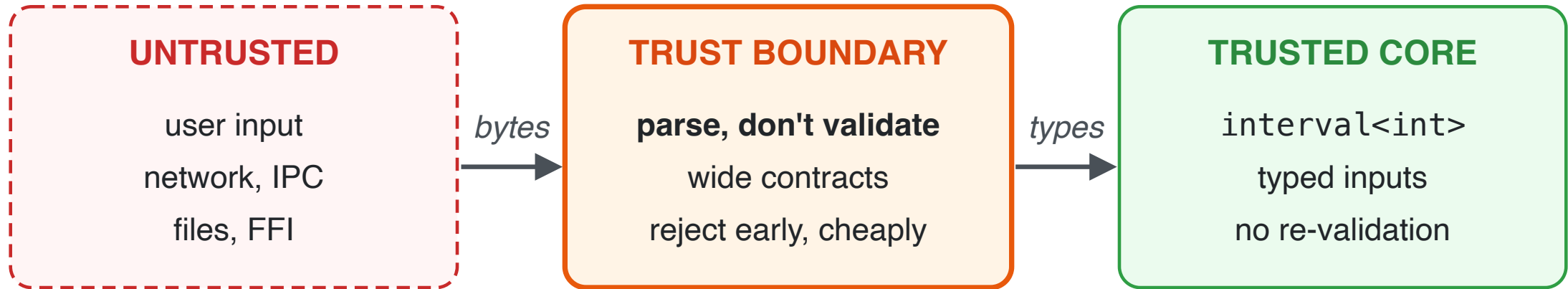
Post-move state: clang-tidy

```
auto r2 = std::move(r);  
use(r.low());  
// bugprone-use-after-move
```

Make it policy, not heroics

- warnings as errors
- sanitizers in CI
- hardened library in test builds
- every suppression has a reason

Check at the trust boundary, trust the types inside



```
// wide: every input gets an answer
std::expected<interval<int>, parse_error> parse_range(std::string_view);

// inside: the type carries the proof, nothing to re-check
int my_clamp(int value, interval<int> const& range);
```

When validity is threatened: contain the blast radius

Constrain the code

- memory-safe components where untrusted bytes arrive
- approved libraries, banned APIs
- static analysis as a CI gate

Constrain the runtime

- least privilege: capabilities
- network: allow-listed endpoints only
- OS: restricted syscalls, sandboxes
- CPU, memory, quotas, timeouts

Example: the image decoder runs in its own process: no network, no file writes.
A bug in it can crash *it*, but not the service.

Terminating stops the damage. Containment also needs **isolation** and **restart-safe state**.

PILLAR 4

Challenge the evidence

Use mutation testing and independent adversarial analysis to challenge claims about correctness.

Specify

Encode

Constrain

Challenge

Govern

✓ A green test suite proves one thing:

The code and the tests agree.

Not that the tests *observe* anything useful.

Not that they probe the *boundaries*.

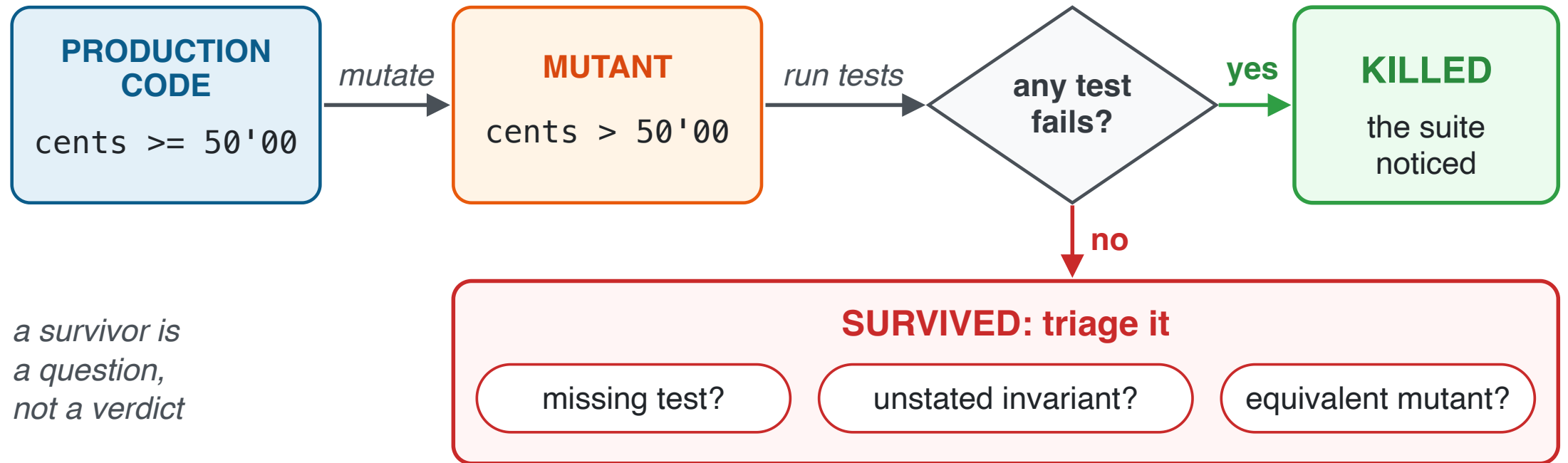
Not that they would notice a *regression*.

Agents are very good at producing a coherent story: an implementation, tests that agree with it, and a confident explanation that it's thoroughly tested.

The same blind spot can live in **both** the code and the tests.

The question that matters: **would our tests fail if the code were wrong?**

Mutation testing in 1 minute



Typical mutants: `<` to `<=`, `&&` to `||`, `+` to `-`, delete a call, `return true`, `return 0`

The boundary nobody tested

```
bool gets_free_shipping(long cents) {  
    return cents >= 50'00;  
}  
  
TEST(Shipping, LargeOrdersShipFree) {  
    EXPECT_TRUE(gets_free_shipping(75'00));  
}
```

Mutant	Result
cents > 50'00	survived
return true;	survived
cents <= 50'00	killed

100% line coverage.

```
EXPECT_TRUE (gets_free_shipping(50'00));    // boundary: kills >  
EXPECT_FALSE(gets_free_shipping(49'99));    // negative: kills return true
```

The new tests don't raise coverage. They **tell the promise apart from a realistic bug.**

Surviving mutants point at unstated invariants

```
if (not (lo_ <= hi_)) throw std::invalid_argument("interval: low > high");
```

Change	A test that tells it apart	What it reveals
delete the <code>throw</code>	<code>{10, 5}</code> must throw	negative cases matter
<code><=</code> becomes <code><</code>	<code>{5, 5}</code> must be accepted	singletons are allowed: is that <i>tested</i> ?
<code>hi_ < lo_</code> instead	<code>int</code> : none. <code>double</code> : <code>{NaN, 1.0}</code>	equivalent, or a missing domain rule

A surviving mutant asks:

- **Is a test missing?**
or
- **Is there an invariant nobody stated?**

Separate authoring from evaluation

Artifact	Produced by	Challenged by
Specification	human intent, AI-assisted drafting	reviewers hunting for gaps and contradictions
Tests	derived from the spec , before the code	mutation testing; held-out negative cases
Implementation	a separate agent; tests read-only	review against the spec; the risk of each change

- **Mutation testing**: a *reproducible* adversary. Same baseline, same survivors
- **Separate roles and contexts**: fewer correlated errors. A different model is *not* a guarantee of independence
- **Humans**: *judgment*. Is the spec what we actually want?

PILLAR 5

Govern the automation

Automate implementation, while humans own intent, acceptance criteria, and operational accountability.

Specify

Encode

Constrain

Challenge

Govern

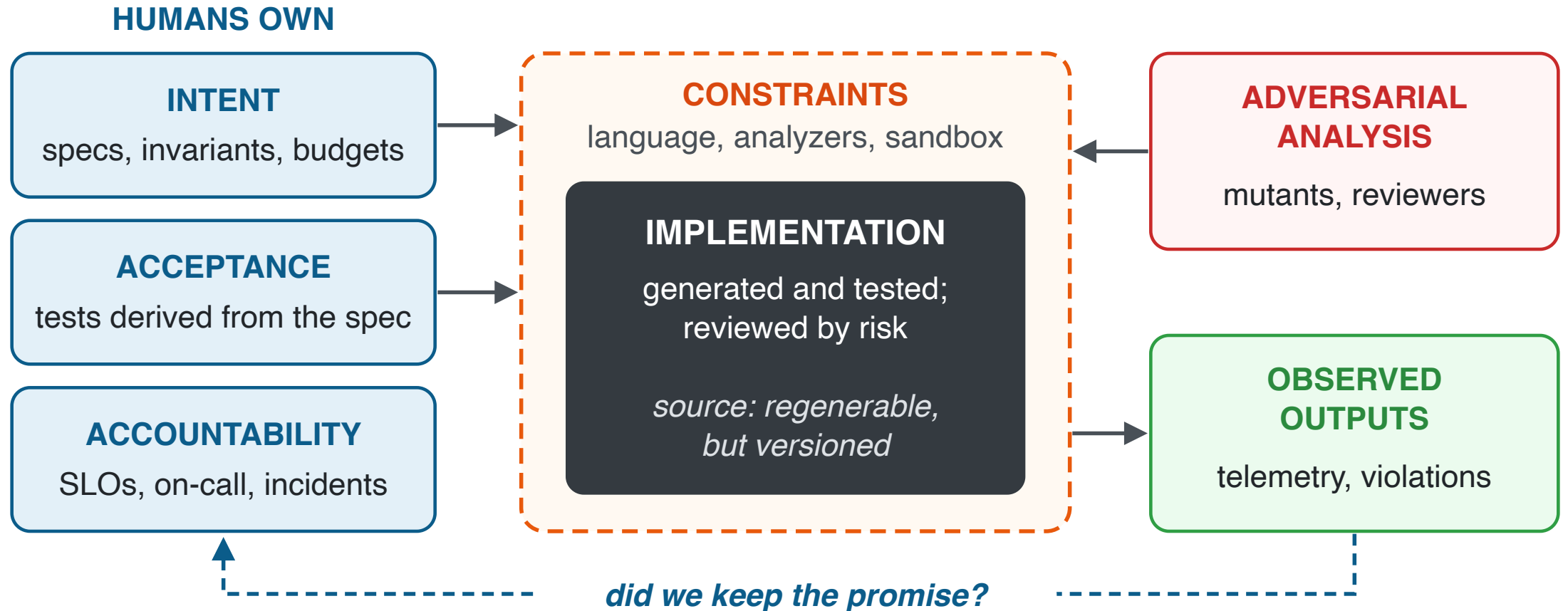
Hot take: Is source code the new "assembly"?

- **Agents** write code faster than any team can review it
- Early compilers produced "slop" too. Then they earned our trust over time
- Unlike a compiler, a model gives **no semantics-preservation guarantee**
- **Abstraction** is key to controlling the "chaos"

If it is, what must **humans** still own in the process?

The invariants: the promise, the acceptance criteria, the accountability.

Humans own the edges of the box



Every invariant needs an Owner

	Local invariants	System invariants
Scope	one type or module	across components and teams
Example	<code>low() <= high()</code>	a payment is never recorded twice
Encoded by	types, constructors, contracts	schemas, protocols, idempotency keys
Enforced by	compiler, analyzers, unit tests	unique keys, idempotent writes, integration tests
Monitored by	contract violations as telemetry	reconciliation, dashboards, alerts
Owned by	the module's maintainers	a named service owner

An invariant nobody owns decays into a comment (at best).

Takeaways

1. **Specify the promise:** pre, post, invariants. Precise enough to challenge, ideally in code
2. **Encode the invariants:** make invalid states hard to represent; make partial states explicit; protect relations, not fields
3. **Constrain the system:** check at the boundary, trust the types inside; unchecked code is small, named, owned
4. **Challenge the evidence:** *coverage* says what ran; *mutants* show what the tests would *catch*
5. **Govern the automation:** implementations may be regenerated. The invariants must be owned

Questions for any code you read:

- What are the invariants?
- Who relies on them?
- How are they stated?
- How are they enforced?
- How would we know if they broke?

Monday morning (at work)

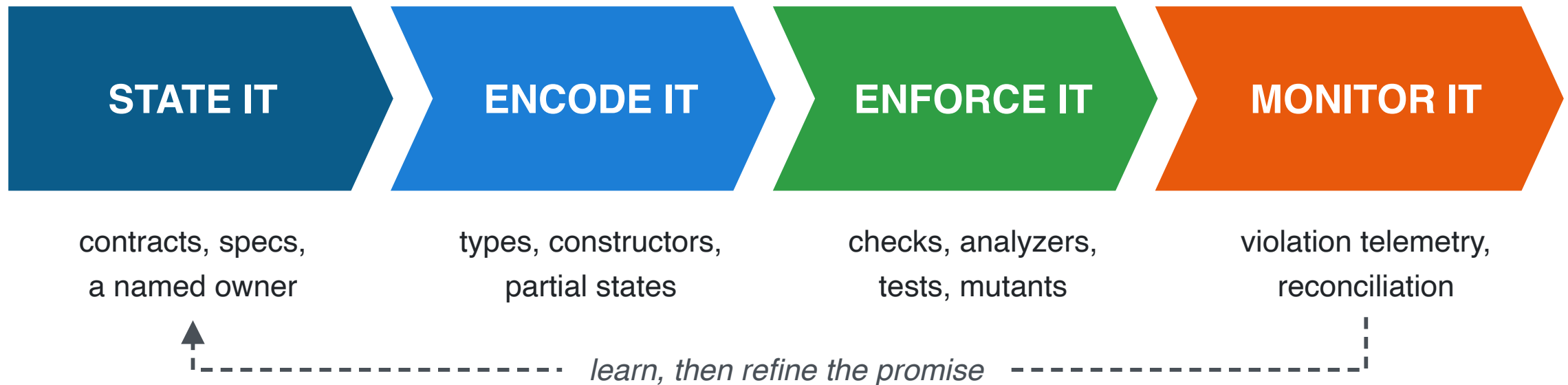
Small pilots, this week

- Pick one component; write down its 3-5 must-never-break invariants
- Turn one comment into `pre` / `post` / `contract_assert`, in a test build
- Turn on the hardened standard library and sanitizers in CI
- Run mutation testing on one critical module; triage every survivor

Go deeper

- Safety vs. validity invariants
- Moved-from and exception-tainted objects; the *minimal* guarantee
- C++26: contracts, library hardening, erroneous behaviour
- Narrow vs. wide contracts
- Mutation testing (e.g. Mull) and property-based testing (e.g. RapidCheck)

Invariants are contracts, not comments.



If you can't state the invariant, you can't claim safety.

Kinds of Invariants

C++ Under the Sea

October 2026 · The Netherlands

Victor Ciura

Principal Architect
Microsoft

Bonus slides

Properties: state it once, let the machine hunt

```
RC_GTEST_PROP(Clamp, Laws, (int a, int b, int v)) {  
    auto [lo, hi] = std::minmax(a, b);  
    interval<int> range{lo, hi};  
    int r = my_clamp(v, range);  
    RC_ASSERT(lo <= r && r <= hi);           // stays inside  
    RC_ASSERT(my_clamp(r, range) == r);      // idempotent  
    if (lo <= v && v <= hi)  
        RC_ASSERT(r == v);                  // identity inside  
}
```

Quiz: find a wrong `my_clamp` that passes all three checks.

Generators hunt for inputs that **falsify** the invariant. Fuzzers do the same at trust boundaries.

Complete the oracle, then challenge it

```
int my_clamp(int v, interval<int> const& range) {  
    if (v < range.low()) return range.low();  
    if (range.high() < v) return range.low();    // one slip: should be high()  
    return v;  
}
```

`my_clamp(10, {1, 5})` returns **1**, not 5. All three checks pass.

```
RC_ASSERT(r == (v < lo ? lo : hi < v ? hi : v));    // the defining law
```

The missing endpoint rule kills it. Derive the oracle from the promise, not from the code.