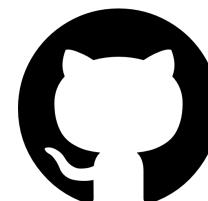


# Tips for writing better (Rust) tests

# Kuba Beránek

- Teaching @ VSB-TUO
- Research @ IT4Innovations
- Co-organizer of Rust meetups :-)
- Open-source contributor @ Rust Project



[github.com/kobzol](https://github.com/kobzol)



# Jakub Beránek

GitHub: [Kobzol](#)

## Team member

- [Bootstrap team](#)
- [Bors team](#)
- [Compiler performance working area](#)
- [Compiler team](#)
- [Crate maintainers](#)
- [Docker team](#)
- [Infrastructure team](#)
- [Leadership council \(\*\*Council Rep Infra\*\*\)](#)
- [Mentors team](#)
- [Mentorship team \(\*\*Lead\*\*\)](#)
- [Parallel rustc working area](#)
- [Rustc Dev Guide working area](#)
- [Survey team \(\*\*Lead\*\*\)](#)
- [Triagebot team](#)

# Leadership council

Charged with the success of the Rust Project as whole, consisting of representatives from top-level teams

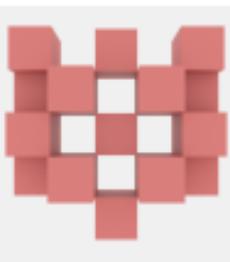
## Members



**Josh Stone**

GitHub: [cuviper](#)

Compiler team



**Eric Huss**

GitHub: [ehuss](#)

Dev tools team



**James Munns**

GitHub: [jamesmunns](#)

Launching pad



**Jakub Beránek**

GitHub: [Kobzol](#)

Infrastructure team



**Mara Bos**

GitHub: [m-ou-se](#)

Library team



**Oliver Scherer**

GitHub: [oli-obk](#)

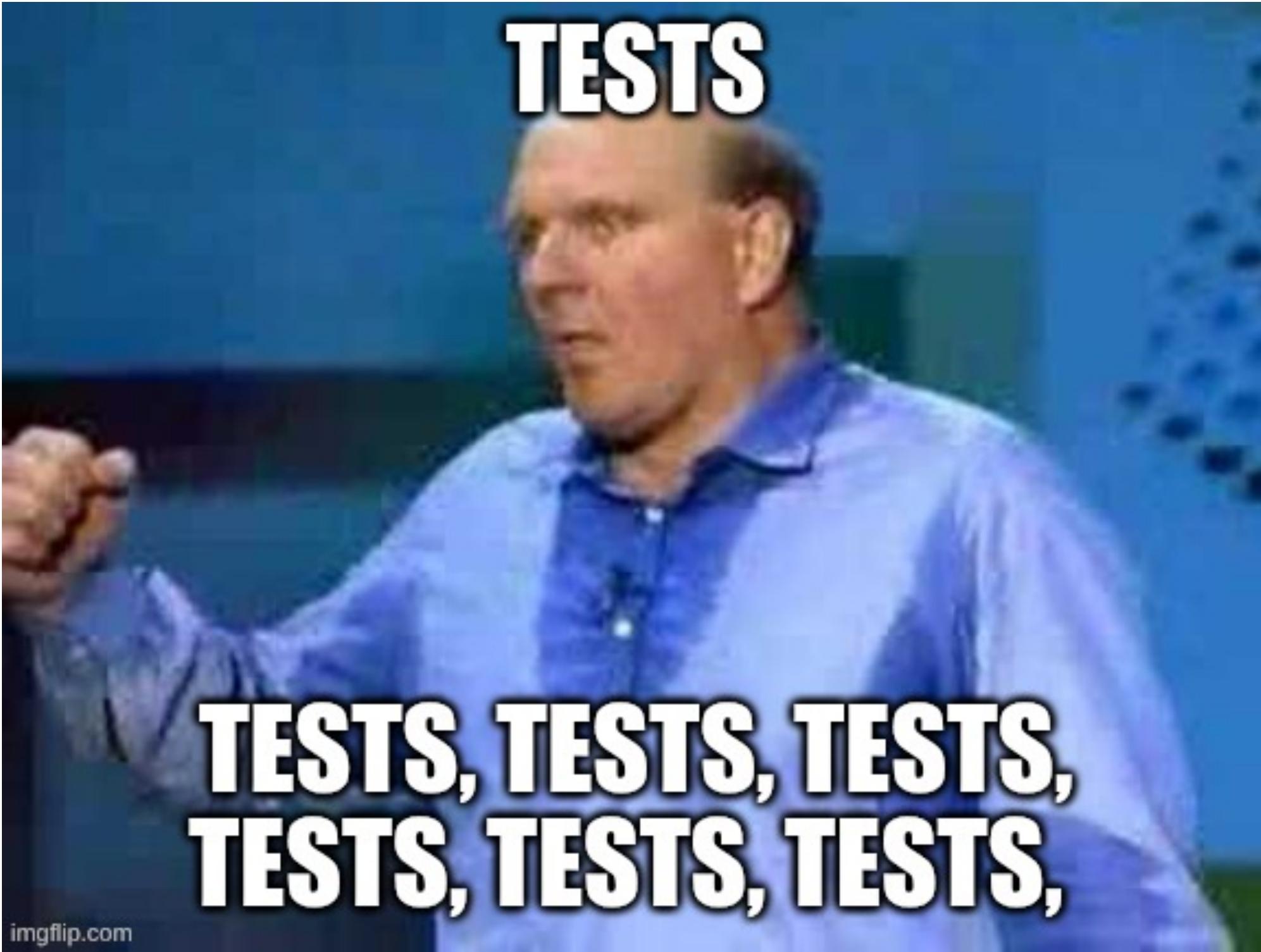
Moderation team



**TC**

GitHub: [traviscross](#)

Language team

A photograph of a man with short, light-colored hair, wearing a blue button-down shirt. He is pointing his right index finger upwards and slightly to the left. The background is a blurred outdoor scene with greenery and a blue sky.

**TESTS**

**TESTS, TESTS, TESTS,  
TESTS, TESTS, TESTS,**

```
$ ./run-tests.sh
```

```
$ ./run-tests.sh
```

```
.....
```

```
$ ./run-tests.sh
```

```
.....
```

```
$ ./run-tests.sh
```

```
.....F.....
```

```
$ ./run-tests.sh
```

```
.....F.....
```

```
$ ./run-tests.sh
```

test result: FAILED. 318 passed; 2 failed;

```
$ ./run-tests.sh
```

```
$ ./run-tests.sh
```

```
.....
```

```
$ ./run-tests.sh
```

```
$ ./run-tests.sh
```

```
.....  
.....
```

```
$ ./run-tests.sh
```

```
.....  
.....  
.....  
.....  
.....
```

```
$ ./run-tests.sh
```

test result: ok. 320 passed; 0 failed;

Why do we bother with tests?

# Why do we bother with tests?

- Find regressions sooner ("shift left")

# Why do we bother with tests?

- Find regressions sooner ("shift left")
- Be more confident when refactoring

# Why do we bother with tests?

- Find regressions sooner ("shift left")
- Be more confident when refactoring
- Help us develop new functionality

# Why do we bother with tests?

- Find regressions sooner ("shift left")
- Be more confident when refactoring
- Help us develop new functionality
- Examples how to use API of our code

# Why do we bother with tests?

- Find regressions sooner ("shift left")
- Be more confident when refactoring
- Help us develop new functionality
- Examples how to use API of our code

Code without (useful) tests => legacy code

Tests kinda suck :-/

Tests kinda suck :-/

- Difficult to maintain

# Tests kinda suck :-/

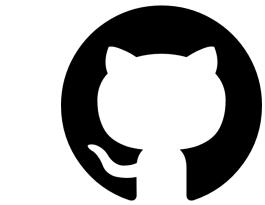
- Difficult to maintain
- Difficult to understand

# Tests kinda suck :-/

- Difficult to maintain
- Difficult to understand
- Difficult to reproduce failures

# Tests kinda suck :-/

- Difficult to maintain
- Difficult to understand
- Difficult to reproduce failures
- Cause stress



**rust-lang/bors**

Merge queue bot

# Disclaimer

- Highly opinionated

# Disclaimer

- Highly opinionated
- Might not apply to your use-cases!

# Disclaimer

- Highly opinionated
- Might not apply to your use-cases!
- Some tips might be obvious

# Disclaimer

- Highly opinionated
- Might not apply to your use-cases!
- Some tips might be obvious  
...others might seem crazy

## Annoyance 1:

Tests break during *refactoring*

1) Do a big refactoring

- 1) Do a big refactoring
- 2) Fix compiler errors

- 1) Do a big refactoring
- 2) Fix compiler errors

3)

```
> cargo build
  Compiling hyperqueue v0.24.0 (/projects/it4i/hyperqueue/crates/hyperqueue)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 3.01s
```



- 1) Do a big refactoring
- 2) Fix compiler errors

3)

```
> cargo build
  Compiling hyperqueue v0.24.0 (/projects/it4i/hyperqueue/crates/hyperqueue)
  Finished `dev` profile [unoptimized + debuginfo] target(s) in 3.01s
```



4)

```
> cargo test
  Compiling hyperqueue v0.24.0 (/projects/it4i/hyperqueue/crates/hyperqueue)
error: could not compile `hyperqueue` (lib test) due to 30 previous errors
```



```
#[test]
fn test_set_priority() {
    let bors = ...;
    let pr = PullRequest {
        repo: "foo/bar".to_string(),
        number: 5,
        author: "bot".to_string()
    };

    let response = bors.post_comment(pr, "@bors priority=5");

    assert_eq!(response.text(), "Priority set to 5");
}
```

**Tip 1:**

Build high-level test APIs

bors

```
#[derive(derive_builder::Builder, Default, Clone)]
pub struct PR {
    #[builder(default = "foo/bar".to_string())]
    repo: String,
    #[builder(default = 1)]
    number: u32,
    ...
}
```

bors

```
#[derive(derive_builder::Builder, Default, Clone)]
pub struct PR {
    #[builder(default = "foo/bar".to_string())]
    repo: String,
    #[builder(default = 1)]
    number: u32,
    ...
}
```

```
#[test]
fn test_set_priority() {
    let bors = ...;
    let pr = PRBuilder::default().author("bot");
    ...
}
```

```
#[bon::builder]
fn create_pr(
    repo: Option<&str>,
    number: Option<u32>
) -> PullRequest {
    ...
}
```

```
bors.post_comment("@bors delegate=try");
bors.get_pr()
    .expect_delegated(DelegatedPermission::Try);

bors.post_comment("@bors r+");
bors.get_pr()
    .expect_approved();
```

Very useful!

```
#[track_caller]
pub fn expect_approved(&self) -> &Self {
    assert!(self.get_pr().is_approved());
    self
}
```

Very useful!

```
#[track_caller]
pub fn expect_approved(&self) -> &Self {
    assert!(self.get_pr().is_approved());
    self
}
```

thread '...' panicked at src/test\_utils.rs:1066:9:

thread '...' panicked at src/bors/handlers/review.rs:560:18:

```
bors.post_comment("foo/bar", 1, "@bors ping");
bors.post_comment("foo/bar", 1, "@bors try");
bors.post_comment("foo/bar", 1, "@bors r+");
bors.post_comment("foo/bar", 2, "@bors ping");
```

```
bors.post_comment("foo/bar", 1, "@bors ping");
bors.post_comment("foo/bar", 1, "@bors try");
bors.post_comment("foo/bar", 1, "@bors r+");
bors.post_comment("foo/bar", 2, "@bors ping");

bors.post_comment("@bors ping");
bors.post_comment_custom("foo/bar", 2, "@bors ping");
```

```
bors.post_comment("foo/bar", 1, "@bors ping");
bors.post_comment("foo/bar", 1, "@bors try");
bors.post_comment("foo/bar", 1, "@bors r+");
bors.post_comment("foo/bar", 2, "@bors ping");

bors.post_comment("@bors ping");
bors.post_comment_custom("foo/bar", 2, "@bors ping");

bors.post_comment(Comment::new("@bors ping").pr(2));
```

```
struct PrIdentifier { repo: Repository, number: u64 }

// The "Into trick":
fn post_comment<Id: Into<PrIdentifier>>(
    &mut self,
    id: Id,
    text: &str
) { ... }
```

```
struct PrIdentifier { repo: Repository, number: u64 }
```

```
struct PrIdentifier { repo: Repository, number: u64 }

impl From<(Repository, u64)> for PrIdentifier {
    fn from((repo, number)): (Repository, u64)) -> Self {
        Self { repo, number }
    }
}
```

```
struct PrIdentifier { repo: Repository, number: u64 }

impl From<(Repository, u64)> for PrIdentifier {
    fn from((repo, number)): (Repository, u64) -> Self {
        Self { repo, number }
    }
}

impl From<u64> for PrIdentifier {
    fn from(number: u64) -> Self {
        Self { repo: default_repo(), number }
    }
}
```

```
struct PrIdentifier { repo: Repository, number: u64 }

impl From<(Repository, u64)> for PrIdentifier {
    fn from((repo, number)): (Repository, u64)) -> Self {
        Self { repo, number }
    }
}

impl From<u64> for PrIdentifier {
    fn from(number: u64) -> Self {
        Self { repo: default_repo(), number }
    }
}

impl From<()> for PrIdentifier {
    fn from(_ : ()) -> Self {
        Self { repo: default_repo(), number: 1 }
    }
}
```

```
// Default repo, default PR  
bors.post_comment((), "@bors ping");
```

```
// Default repo, default PR  
bors.post_comment((), "@bors ping");
```

```
// Default repo, PR 2  
bors.post_comment(2, "@bors ping");
```

```
// Default repo, default PR  
bors.post_comment((), "@bors ping");
```

```
// Default repo, PR 2  
bors.post_comment(2, "@bors ping");
```

```
// Repo foo/bar, PR 2  
bors.post_comment(("foo/bar", 2), "@bors ping");
```

```
// Default repo, default PR
bors.post_comment((), "@bors ping");

// Default repo, PR 2
bors.post_comment(2, "@bors ping");

// Repo foo/bar, PR 2
bors.post_comment(("foo/bar", 2), "@bors ping");

// Custom comment author
bors.post_comment(("foo/bar", 2),
    Comment::new(User::new("user1"), "@bors ping")
);
```

# Code reviewers:



**DUPLICATED  
CODE**



**DUPLICATED  
CODE IN TESTS**

## Tip 2:

Test public (rather than private) interfaces

## Tip 2:

Test public (rather than private) interfaces  
a.k.a. Test behavior, not implementation



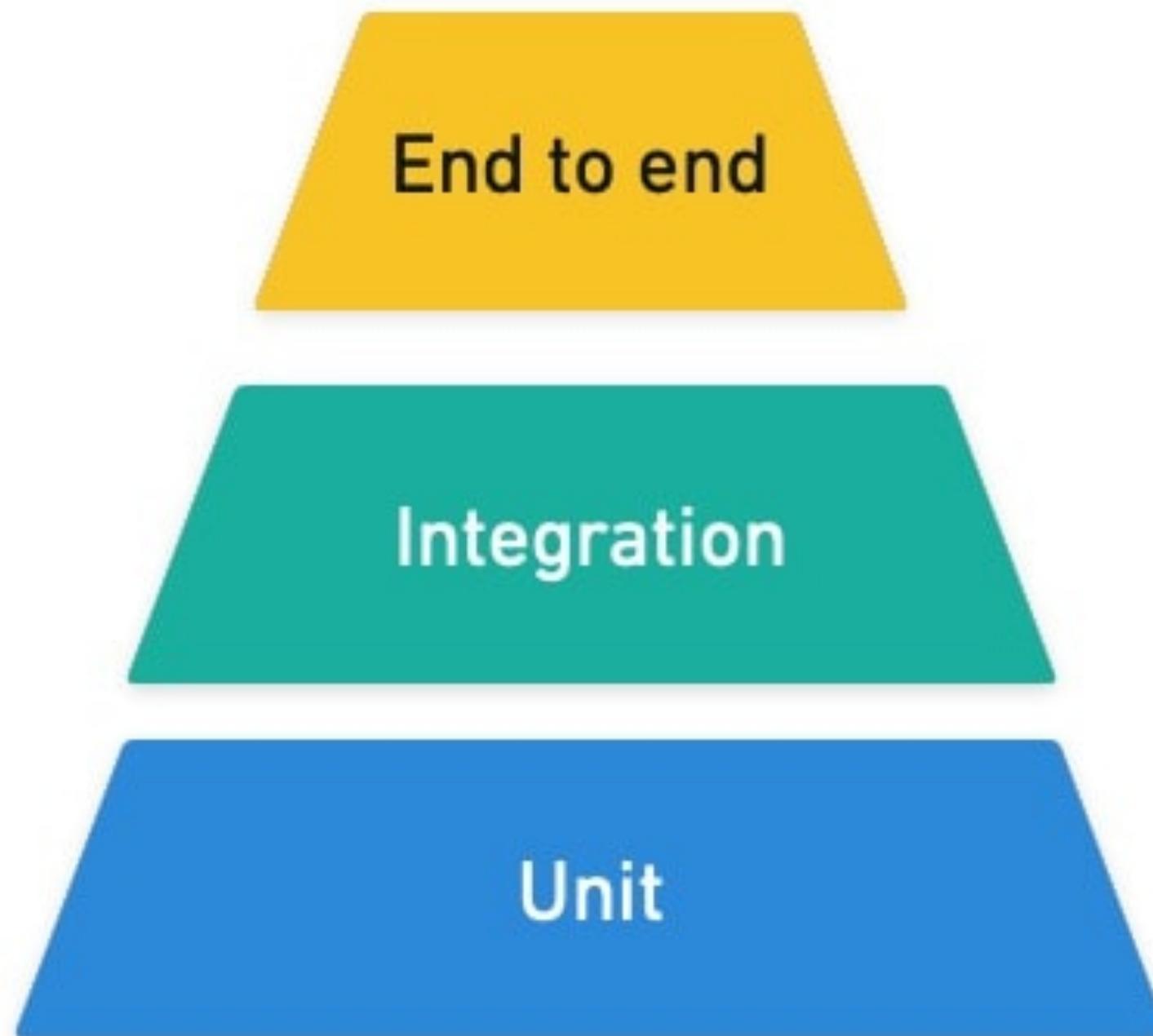
## Tip 2:

Test public (rather than private) interfaces

a.k.a. Test behavior, not implementation

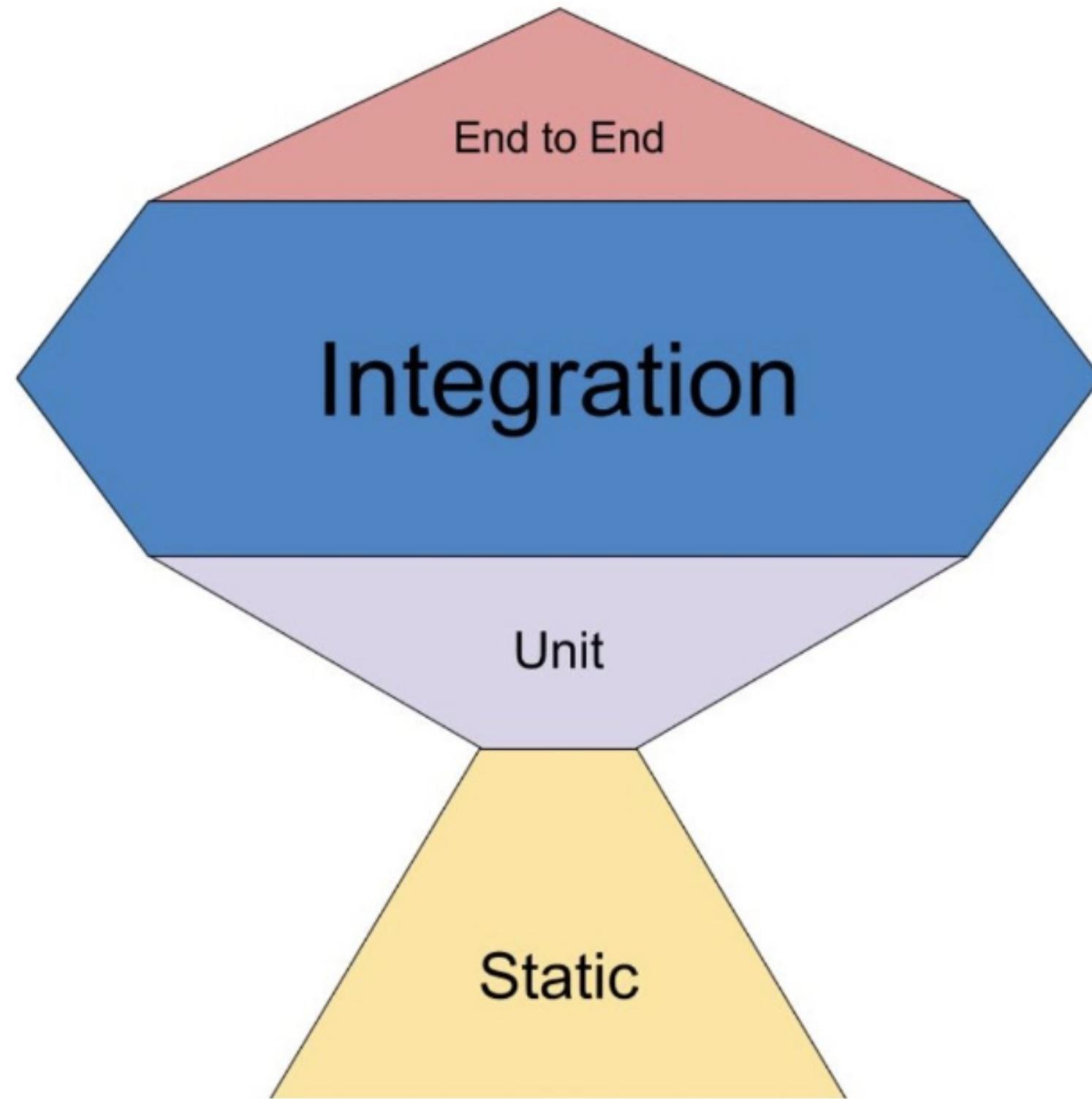
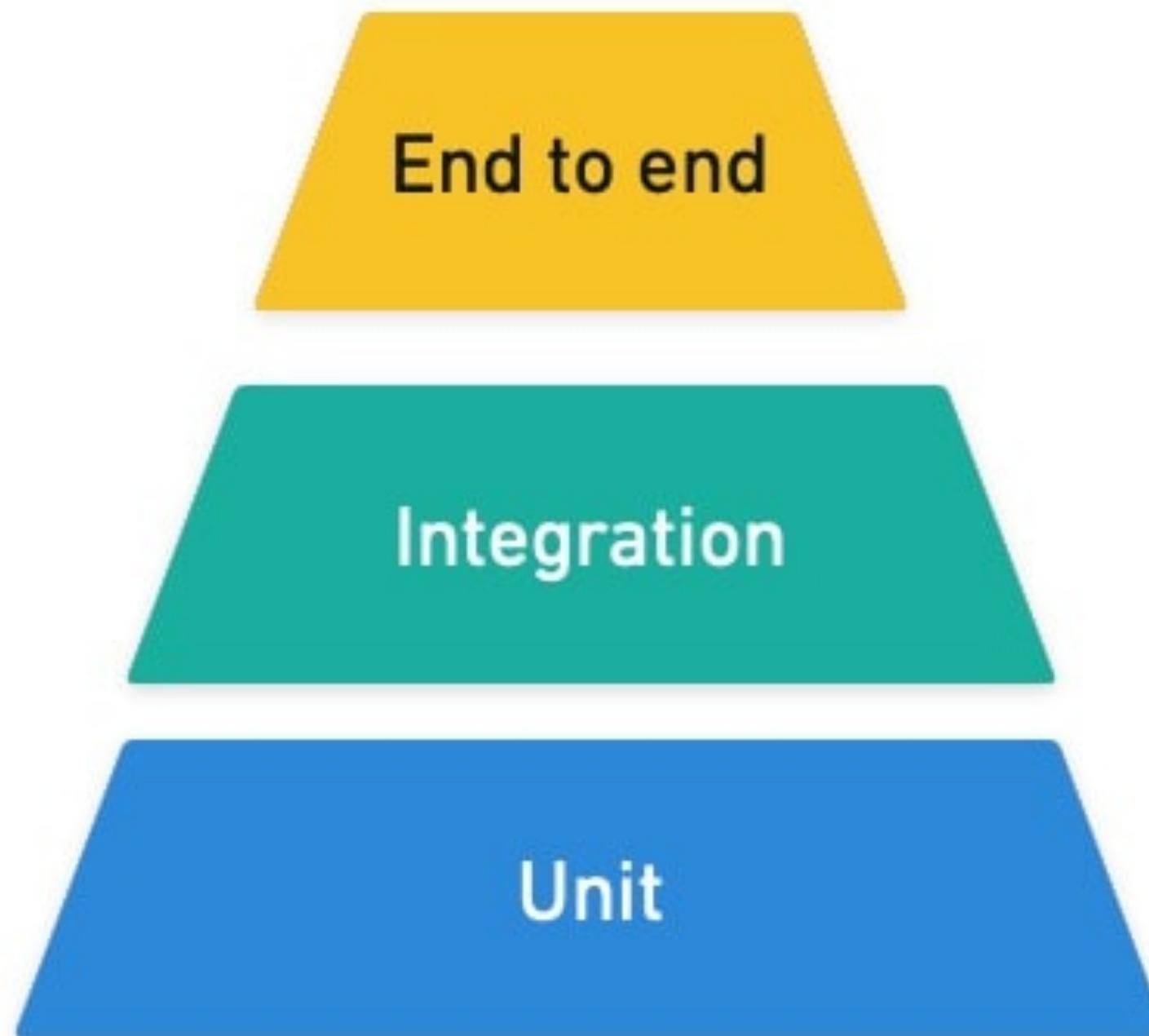
a.k.a. Prefer black-box (rather than white-box) tests

# The Test Pyramid



Source: semaphore.io, Kent C. Dodds

# The Test Pyramid



Source: semaphore.io, Kent C. Dodds

# Neural Network Test

“

Can you re-use the test suite if your entire software is replaced with an opaque neural network?

Aleksey Kladov (matklad) ,”



```
#[test]
fn test_try_build() {
    let bors = // create test context
    bors.post_comment("@bors try");
    let comment = bors.get_comment();
    assert_eq!(comment, "Try build started");

}
```

```
#[test]
fn test_try_build() {
    let bors = // create test context
    bors.post_comment("@bors try");
    let comment = bors.get_comment();
    assert_eq!(comment, "Try build started");

    let ci_workflow = Workflow::from(bors.try_branch());
    bors.workflow_start(&ci_workflow);
    bors.workflow_success(&ci_workflow);

}
```



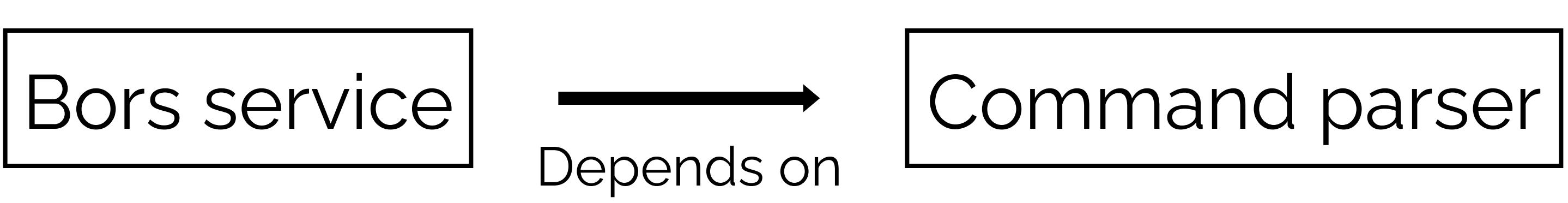
```
#[test]
fn test_try_build() {
    let bors = // create test context
    bors.post_comment("@bors try");
    let comment = bors.get_comment();
    assert_eq!(comment, "Try build started");

    let ci_workflow = Workflow::from(bors.try_branch());
    bors.workflow_start(&ci_workflow);
    bors.workflow_success(&ci_workflow);

    let comment = bors.get_comment();
    assert_eq!(comment, "Try build successful");
}
```



```
#[test]
fn parse_default_approve() {
    let cmd = parse_commands("@bors r+");
    assert_eq!(cmd, vec![
        Ok(BorsCommand::Approve {
            approver: Approver::Myself,
            priority: None,
        })
    ]);
}
```

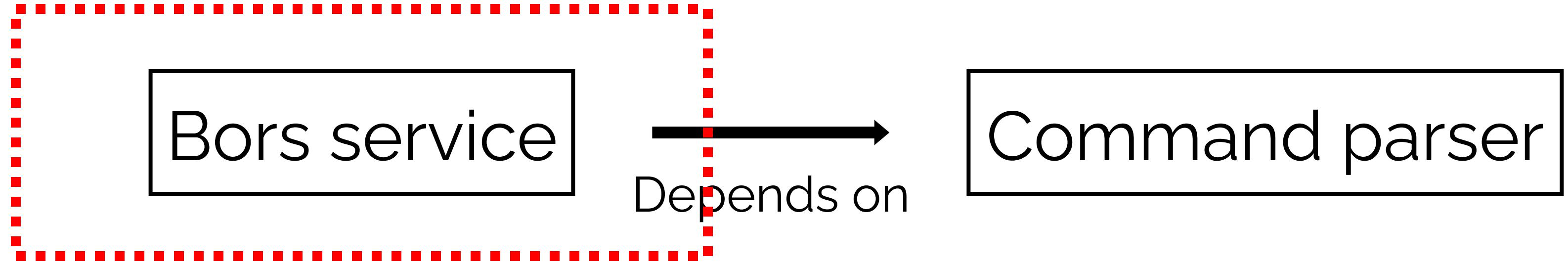


Bors service

→  
Depends on

Command parser

Parser tests



Service tests



Bors service

→  
Depends on

Command parser

Service tests

## Annoyance 2:

Tests are hard to understand

```
let mut buf = VecDeque::new();
```

```
let mut buf = VecDeque::new();
```

```
buf.push_back(5);  
buf.push_front(3);  
buf.push_back(4);
```

```
let mut buf = VecDeque::new();

buf.push_back(5);
buf.push_front(3);
buf.push_back(4);

let c: Vec<&i32> = buf.iter().collect();
eprintln!("{}:?}", c); // [3, 5, 4]
```



In 2022:

```
let v = Vec::new();           // no allocation
```



In 2022:

```
let v = Vec::new();           // no allocation
let m = HashMap::new();     // no allocation
```



In 2022:

```
let v = Vec::new();           // no allocation
let m = HashMap::new();     // no allocation
let d = VecDeque::new();    // allocated :(
```

# Improve `VecDeque` implementation #99805

✓ Closed



joboet opened on Jul 27, 2022

Member

...

`VecDeque` currently allocates one extra empty element because it needs to discern between an empty and full queue. Besides wasting memory, this means `VecDeque::new` cannot currently be `const`.

## Solution

The most elegant solution would be to reimplement `VecDeque` based off the (third) technique described by Juho Snellman [in this blog post](#). In this implementation, the buffer indexes are not clamped to the buffer size, but instead use the whole range of `usize`.

```
#[test]
fn test_grow_full_middle_copy_after_t_2() {
    let mut vd = VecDeque::<u64>::with_capacity(4);
    vd.push_back(5);
    vd.pop_front();
    vd.push_back(1);
    vd.push_back(2);
    vd.push_back(3);
    vd.push_back(4);

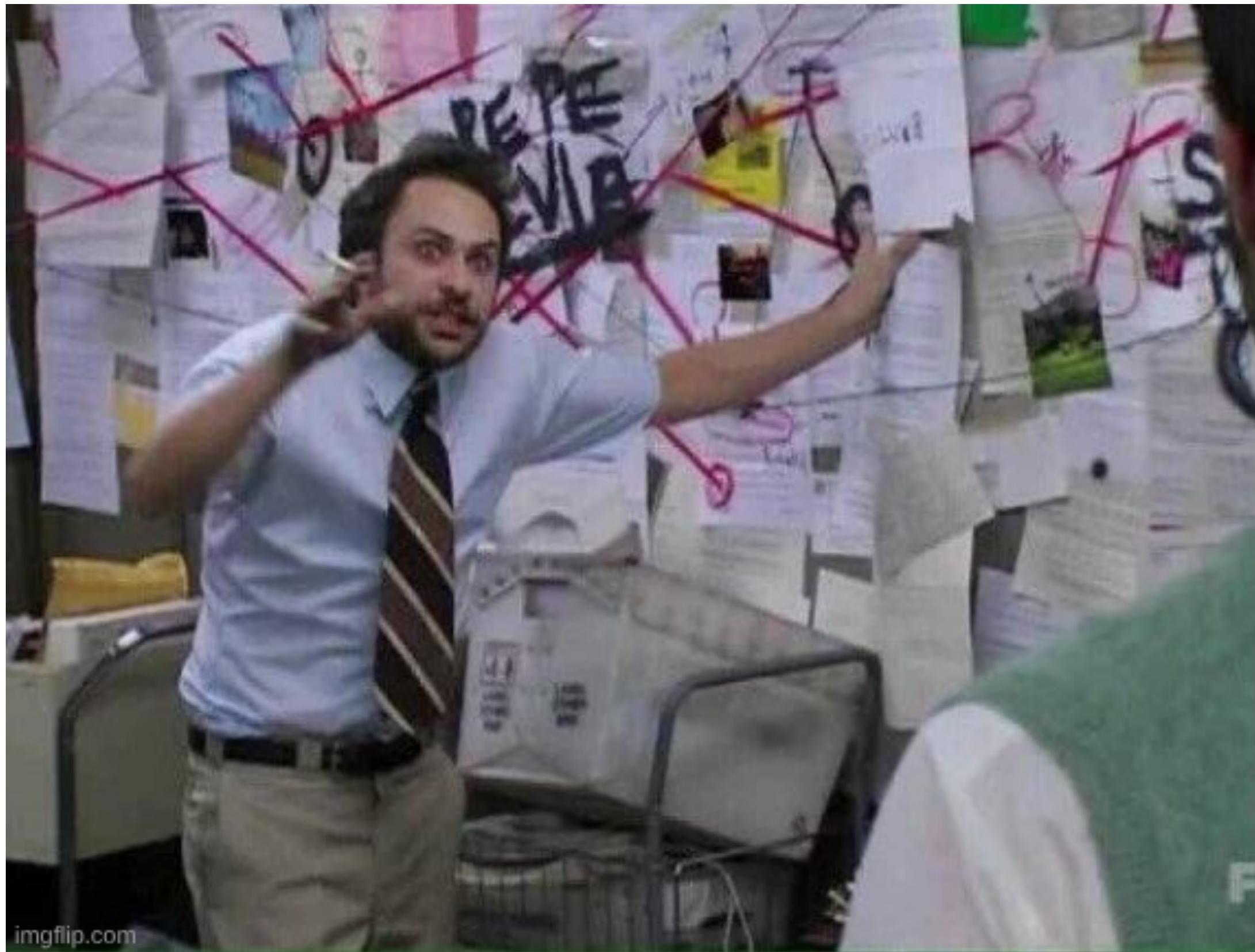
    assert_eq!(vd.tail_index(), 1);
    assert_eq!(vd.head_index(), 1);
    assert_eq!(vd.phantom_head_index(), 5);
    assert_eq!(vd.get(0), Some(4));

    vd.reserve_exact(vd.capacity()); // shrink capacity

    assert_eq!(vd.tail_index(), 1);
    assert_eq!(vd.head_index(), 5);
    assert_eq!(vd.get(4), Some(4));
}
```



Me trying to understand complex tests:





**Tip 3:**

Make tests visual



### Tip 3:

Make tests visual  
to make them easier to understand

```
#[test]
fn test_grow_full_middle_copy_after_t_2() {
    let mut vd = VecDeque::<u64>::with_capacity(4);
    ...
    assert_eq!(render(&vd), r#"4,1,2,3|_,_,_
t      H
h
"#);
    vd.reserve_exact(vd.capacity()); // shrink capacity
    assert_eq!(render(&vd), r#"_,1,2,3,4,_,_,_|_,_,_,_,_,_,_
t          H
"#);
}
}
```



Be creative!



Imagine you're building an IDE for Rust



Imagine you're building an IDE for Rust

How would you write a test for "Go to definition"?



```
fun `test type alias`() = checkByCode(  
    type Foo = usize;  
    //X  
  
    trait T { type 0; }  
  
    struct S;  
  
    impl T for S { type 0 = Foo; }  
    //^  
    """)
```

# Rust Analyzer



```
#[test]
fn goto_definition_resolves_correct_name() {
    check(r#"
//- /lib.rs
use a::Foo;
mod a;
mod b;

enum E { X(Foo$0) }

// - /a.rs
struct Foo;
    //^^^

// - /b.rs
struct Foo;
"#,
    );
}
```

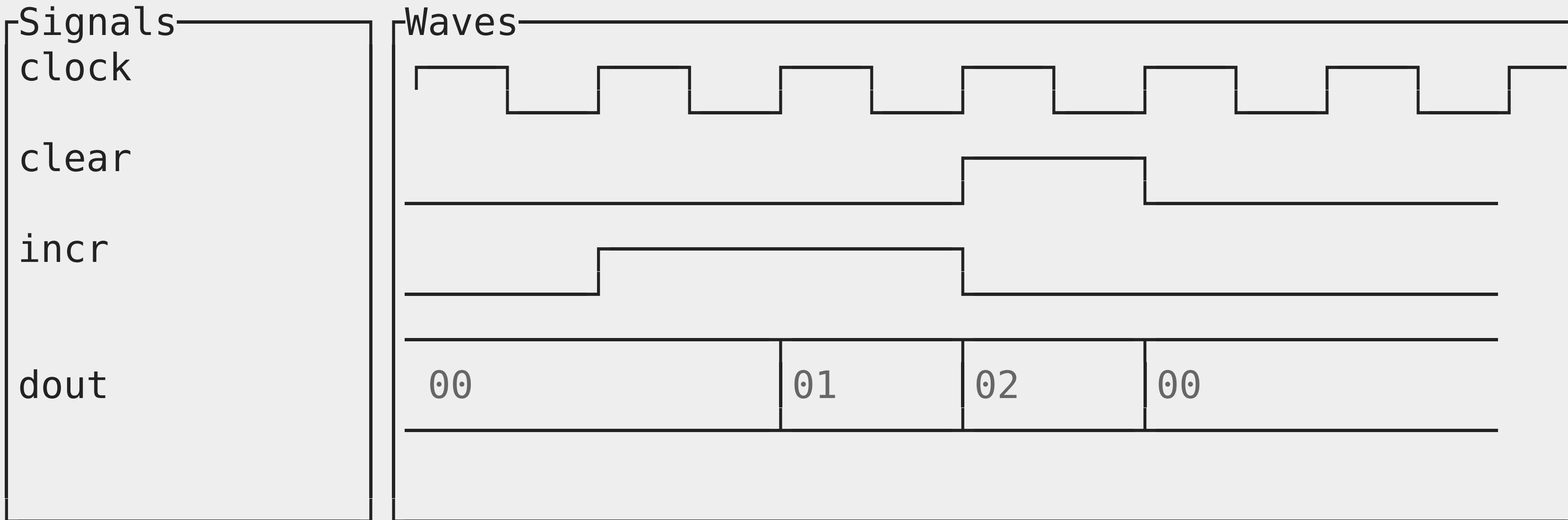
# bootstrap (Rust compiler build system)

```
#[test]
fn cross_compile_library_stage_2() {
    let ctx = TestCtx::new();
    assert_eq!(
        ctx.config("build")
            .path("library")
            .stage(2)
            .targets(&["target1"])
            .render_steps(), r#
        [build] llvm <host>
        [build] rustc 0 <host> -> rustc 1 <host>
        [build] rustc 1 <host> -> std 1 <host>
        [build] rustc 1 <host> -> rustc 2 <host>
        [build] rustc 2 <host> -> std 2 <host>
        [build] rustc 1 <host> -> std 1 <target1>
        [build] rustc 2 <host> -> std 2 <target1>
    "#);
}
```

# Hardcaml test suite



```
let waves = testbench ();;
val waves : Waveform.t = <abstr>
Waveform.print ~display_height:12 waves;;
```



```
- : unit = ()
```



## Annoyance 3:

It is hard to adapt tests to new behavior



**Tip 4:**

Make tests data-driven



## Tip 4:

Make tests data-driven  
a.k.a. Make tests blesstable



## Tip 4:

Make tests data-driven

a.k.a. Make tests blesstable

a.k.a. Use snapshot/expect tests

# bootstrap (Rust compiler build system)

```
#[test]
fn cross_compile_library_stage_2() {
    let ctx = TestCtx::new();
    assert_eq!(
        ctx.config("build")
            .path("library")
            .stage(2)
            .targets(&["target1"])
            .render_steps(), r#
        [build] llvm <host>
        [build] rustc 0 <host> -> rustc 1 <host>
        [build] rustc 1 <host> -> std 1 <host>
        [build] rustc 1 <host> -> rustc 2 <host>
        [build] rustc 2 <host> -> std 2 <host>
        [build] rustc 1 <host> -> std 1 <target1>
        [build] rustc 2 <host> -> std 2 <target1>
    "#);
}
```

# bootstrap (Rust compiler build system)

```
#[test]
fn cross_compile_library_stage_2() {
    let ctx = TestCtx::new();
    insta::assert_snapshot!(
        ctx.config("build")
            .path("library")
            .stage(2)
            .targets(&["target1"])
            .render_steps(), @r#"
[build] llvm <host>
[build] rustc 0 <host> -> rustc 1 <host>
[build] rustc 1 <host> -> std 1 <host>
[build] rustc 1 <host> -> rustc 2 <host>
[build] rustc 2 <host> -> std 2 <host>
[build] rustc 1 <host> -> std 1 <target1>
[build] rustc 2 <host> -> std 2 <target1>
"#);
}
```

```
$ cargo insta review
```



# \$ cargo insta review

Reviewing [1/1] bootstrap@0.0.0:

Snapshot: check\_library\_stage\_2

Source: /projects/personal/rust/rust/src/bootstrap:1926

Expression: ctx.config("check").path("library").stage(2).render\_steps()

-old snapshot

+new results

0	0	[build] llvm <host>
1	1	[build] rustc 0 <host> -> rustc 1 <host>
2	2	[build] rustc 1 <host> -> std 1 <host>
3	3	+ [build] rustc 1 <host> -> rustc 2 <host>
3	4	[check] rustc 2 <host> -> std 2 <host>

**a** accept

keep the new snapshot

**r** reject

retain the old snapshot

**s** skip

keep both for now

**i** hide info

toggles extended snapshot info

**d** hide diff

toggle snapshot diff



```
bors
    .post_comment("@bors try parent=last")
    .await?;

insta::assert_snapshot!(
    bors.get_next_comment_text(()).await?,
    @":exclamation: There was no previous build.
Please set an explicit parent or remove the
`parent=last` argument to use the default parent."
);
```

# Rust compiler UI test suite



```
= array-slice-coercion-mismatch-15/a  
≡ assign-imm-local-twice.fixed  
② assign-imm-local-twice.rs  
≡ assign-imm-local-twice.stderr  
② assign-never-type.rs
```



```
//@ run-rustfix

fn main() {
    let v: isize;
    //~^ HELP consider making this binding mutable
    //~| SUGGESTION mut
    v = 1;
    //~^ NOTE first assignment
    println!("v={}", v);
    v = 2;
    //~^ ERROR cannot assign twice to immutable variable
    //~| NOTE cannot assign twice to immutable
    println!("v={}", v);
}
```



```
error[E0384]: cannot assign twice to immutable variable `v`
--> $DIR/assign-imm-local-twice.rs:17:5
```

```
LL |     v = 1;
|     ----- first assignment to `v`
```

```
...
LL |     v = 2;
|     ^^^^^ cannot assign twice to immutable variable
```

```
help: consider making this binding mutable
```

```
LL | let mut v: isize;
|     +++
```

```
error: aborting due to 1 previous error
```

```
For more information about this error, try `rustc --explain E0384`.
```



```
//@ run-rustfix

fn main() {
    let mut v: isize;
    //~^ HELP consider making this binding mutable
    //~| SUGGESTION mut
    v = 1;
    //~^ NOTE first assignment
    println!("v={}", v);
    v = 2;
    //~^ ERROR cannot assign twice to immutable variable
    //~| NOTE cannot assign twice to immutable
    println!("v={}", v);
}
```



tests/ui/error-codes/E0121.stderr

@@ -2,10 +2,13 @@ error[E0121]: the placeholder `\_` is not allowed within types on item signatures

2 - -> \$DIR/E0121.rs:1:13

3 |

4 LL | fn foo() -> \_ { 5 }

5 - | ^ not allowed in type signatures

6 - | |

7 - | help: replace with the correct return type

8 - | help: replace with the correct return type: `i32`

9

10 error[E0121]: the placeholder `\_` is not allowed within types on item signatures for static variables

11 --> \$DIR/E0121.rs:3:13

....

2 - -> \$DIR/E0121.rs:1:13

3 |

4 LL | fn foo() -> \_ { 5 }

5 + | ^ not allowed in type signatures

6 + |

7 + help: replace with the correct return type

8 + |

9 + LL - fn foo() -> \_ { 5 }

10 + LL + fn foo() -> i32 { 5 }

11 + |

12

13 error[E0121]: the placeholder `\_` is not allowed within types on item signatures for static variables

14 --> \$DIR/E0121.rs:3:13

# Cargo



```
#[cargo_test]
fn build_lib_only() {
    let p = project()
        .file("src/main.rs", "fn main() {}")
        .file("src/lib.rs", r#" "#)
        .build();

    p.cargo("build --lib -v")
        .with_stderr_data(str![["r#"
[COMPILING] foo v0.0.1 ([ROOT]/foo)
[RUNNING] `rustc --crate-name foo --edition=2015 src/lib.rs [..]
--crate-type lib --emit=[..]link[..]
-L dependency=[ROOT]/foo/target/debug/deps`
[FINISHED] `dev` profile [unoptimized + debuginfo] target(s) in [ELAPSED]s
"#]])
        .run();
}
```



## Annoyance 4:

Green tests, red production



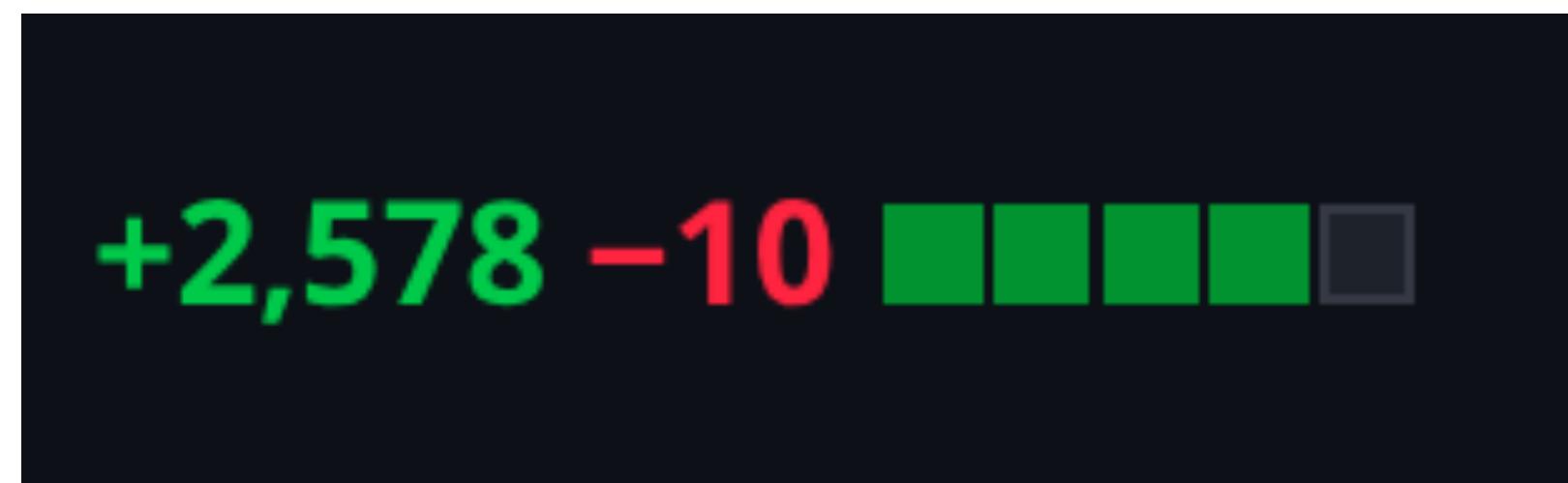
## Tip 5:

Test the *real thing*



## Tip 5:

Test the *real thing*  
a.k.a. Don't (over)use mocks





```
@pytest.mark.asyncio
▶ async def test_get_current_image_id_success(manager_instance):
    mock_container = MagicMock()
    mock_container.image.id = "sha256:imgid"
    manager_instance.client.containers.get.return_value = mock_container
    assert manager_instance._get_current_image_tag() == "imgid"

@ pytest.mark.asyncio
▶ async def test_pull_new_image_already_exists(manager_instance):
    mock_image = MagicMock()
    manager_instance.client.images.get.return_value = mock_image
    manager_instance._pull_new_image()
    manager_instance.client.images.get.assert_called_once_with(manager_instance.new_image)
```





```
struct BorsService {  
    db: sqlx::PgPool  
}
```



```
struct BorsService {
    db: sqlx::PgPool
}

#[test]
fn bors_service() {
    let service = BorsService {
        db: ???
    };
    ...
}
```



```
trait Database { ... }
```



```
trait Database { ... }

struct BorsService<DB: Database> {
    db: DB
}
```



```
trait Database { ... }
```

```
struct BorsService<DB: Database> {
    db: DB
}
```

```
fn run_service<DB: Database>(service: BorsService<DB>) {}
```

```
trait Database { ... }
```

```
struct BorsService<DB: Database> {
    db: DB
}
```

```
fn run_service<DB: Database>(service: BorsService<DB>) {}

fn start_workflow<DB: Database>(
    service: BorsService<DB>,
    id: WorkflowId
) {}
```



```
trait Database { ... }

struct BorsService {
    db: Box<dyn Database>
}
```



```
struct InMemoryDb {  
    pull_requests: HashMap<u32, PullRequest>  
}
```



```
struct InMemoryDb {  
    pull_requests: HashMap<u32, PullRequest>  
}  
impl Database for InMemoryDb { ... }
```



```
struct InMemoryDb {
    pull_requests: HashMap<u32, PullRequest>
}
impl Database for InMemoryDb { ... }

#[test]
fn test_bors() {

    let service = BorsService {
        db: InMemoryDb::default()
    };
    ...
}
```



```
struct InMemoryDb {
    pull_requests: Mutex<HashMap<u32, PullRequest>>
}
impl Database for Arc<InMemoryDb> { ... }

#[test]
fn test_bors() {
    let db = Arc::new(InMemoryDb::default());
    let service = BorsService {
        db: db.clone()
    };
    ...
    assert!(db.pull_requests().lock()...);
}
```



```
Error: error serializing parameter 1
```

```
Caused by:
```

```
    cannot convert between the Rust type `u32` and the Postgres type `int4`
```



```
#[sqlx::test]
async fn bors_service(db: PgPool) {
    let bors = BorsService {
        db
    };
    ...
}
```



Isn't that slow?



Isn't that slow?

~250 integration tests in bors:

- ~15s (debug)
- ~8s (release)



```
services:  
  db:  
    image: postgres:16.9
```



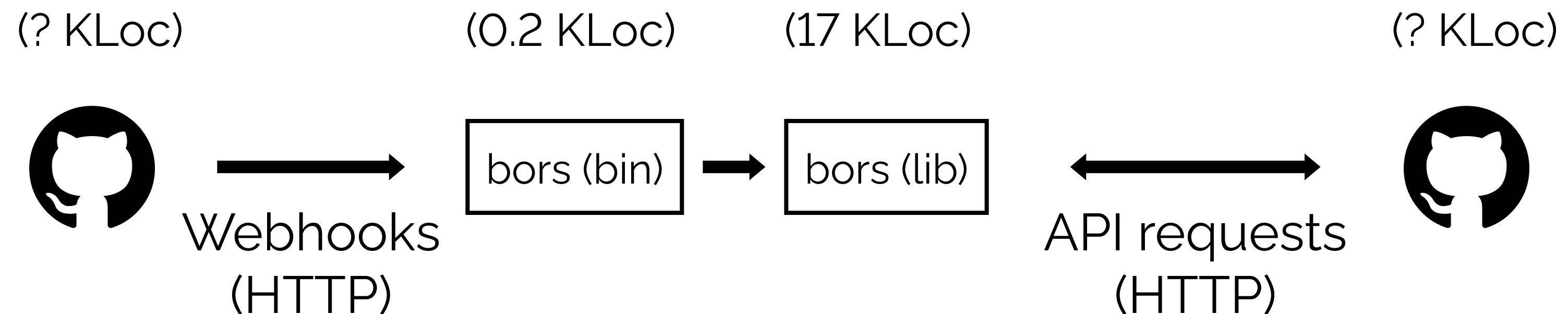
```
services:  
  db:  
    image: postgres:16.9  
    # Turn off durability  
    command: -c fsync=off
```

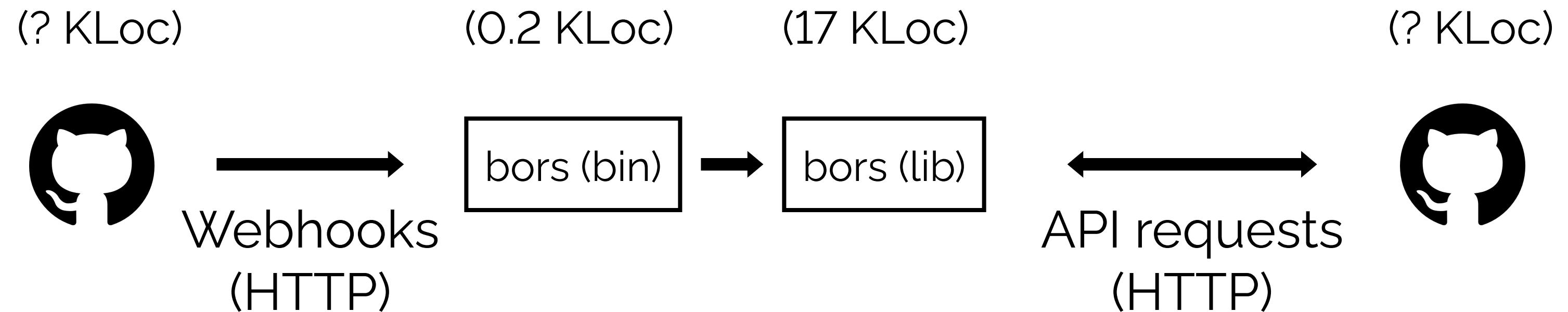


```
services:  
  db:  
    image: postgres:16.9  
    # Turn off durability  
    command: -c fsync=off  
    # Use RAMdisk for storage  
    tmpfs:  
      - /var/lib/postgresql/data
```

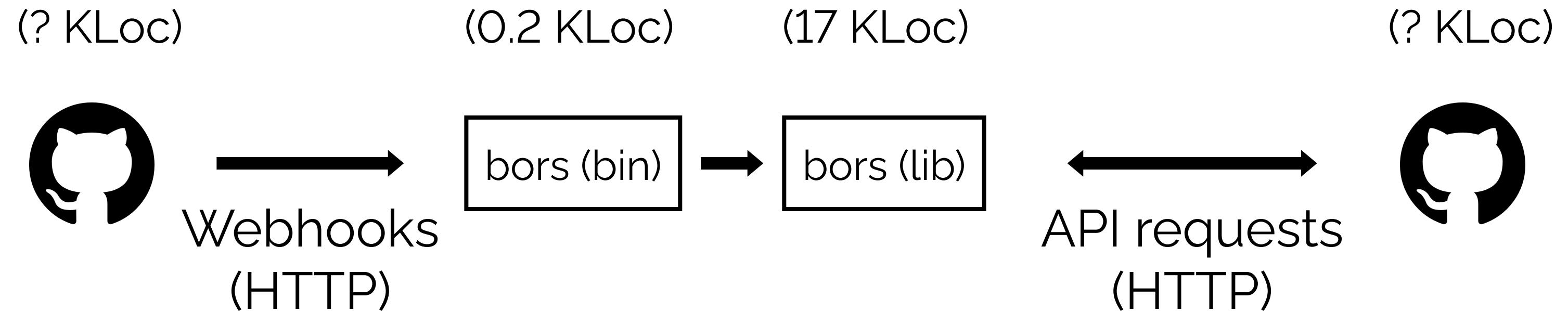


Be pragmatic!

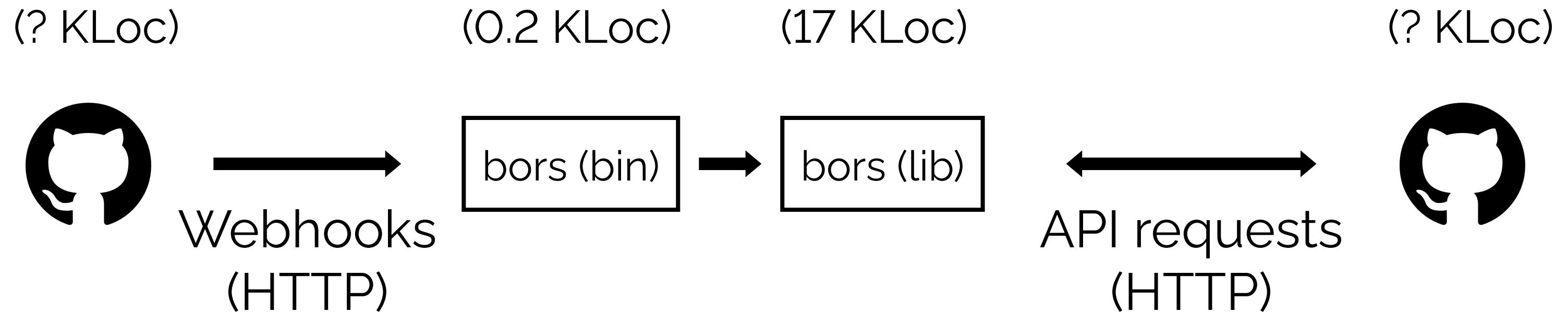




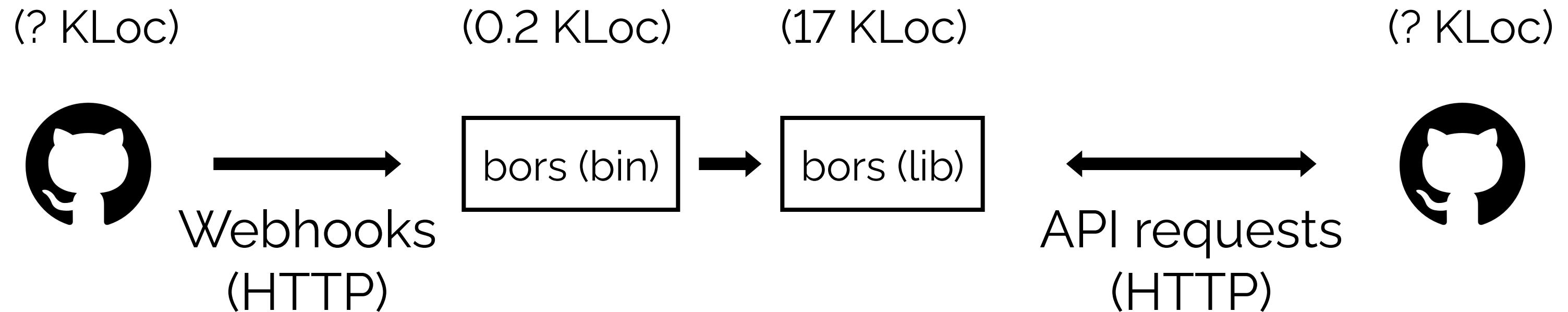
- Reimplement GitHub



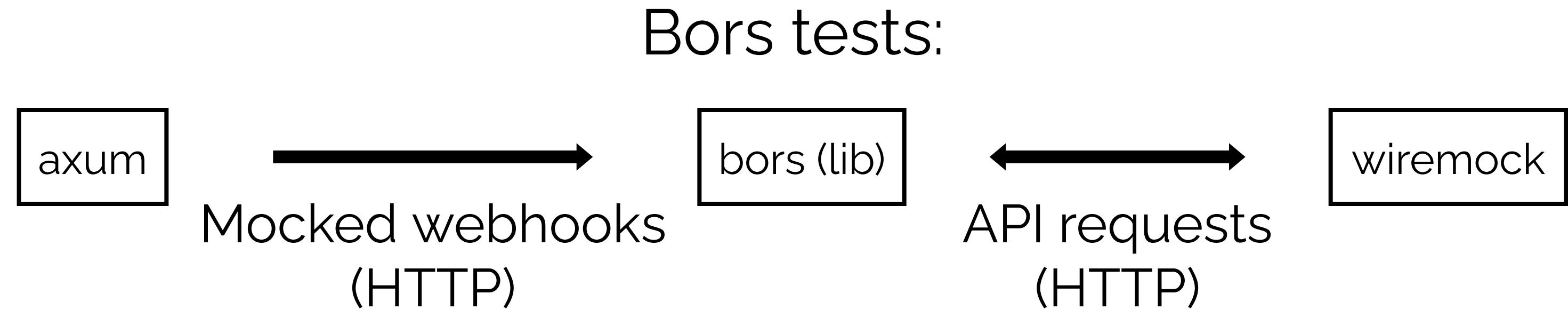
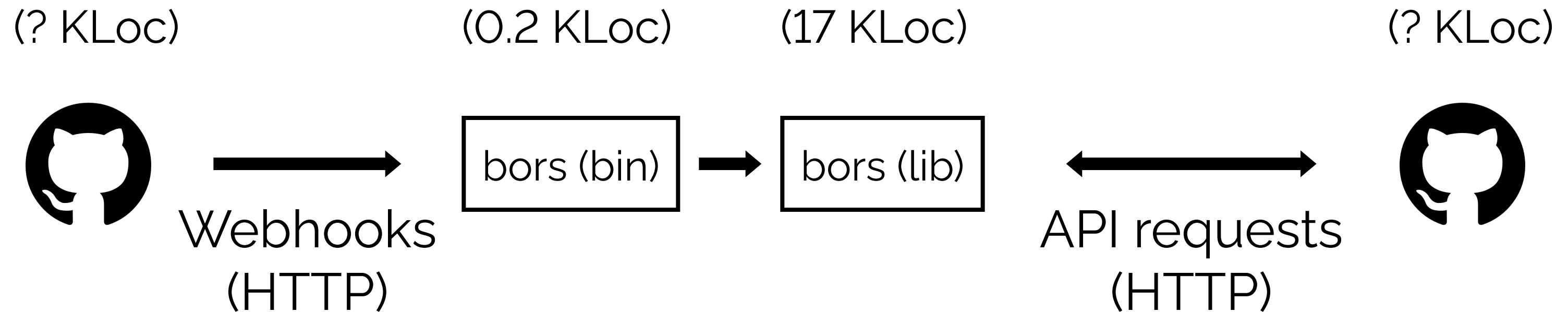
- Reimplement GitHub
- Deploy GitHub locally



- Reimplement GitHub
- Deploy GitHub locally
- Test on real GitHub repositories



- Reimplement GitHub
- Deploy GitHub locally
- Test on real GitHub repositories
- Mock HTTP GitHub endpoints





```
Mock::given(method("POST"))
    .and(path_regex("^/app/installations/\d+/access_tokens$"))
    .respond_with(
        ResponseTemplate::new(200)
            .set_body_json(InstallationToken::default())
    )
    .mount(mock_server)
    .await;
```

```
Mock::given(method("GET"))
    .and(path_regex(format!("^/repos/{repo_name}/pulls/([0-9]+)$")))
    .respond_with(move |req: &Request| {
        // ...
    })
    .mount(mock_server)
    .await;
```



```
Mock::given(method("GET"))
    .and(path_regex(format!("^/repos/{repo_name}/pulls/([0-9]+)$")))
    .respond_with(move |req: &Request| {
        let num = req.url.path_segments().unwrap().next_back().unwrap();
        let pr_number: u64 = num.parse().unwrap();

    })
    .mount(mock_server)
    .await;
```

```
Mock::given(method("GET"))
    .and(path_regex(format!("^/repos/{repo_name}/pulls/([0-9]+)$")))
    .respond_with(move |req: &Request| {
        let num = req.url.path_segments().unwrap().next_back().unwrap();
        let pr_number: u64 = num.parse().unwrap();
        let pull_request_error = repo.lock().error;
        if pull_request_error {
            ResponseTemplate::new(500)
        } else if let Some(pr) = repo.lock().prs.get(&pr_number) {
            ResponseTemplate::new(200).set_body_json(
                GitHubPullRequest::from(pr)
            )
        } else {
            ResponseTemplate::new(404)
        }
    })
    .mount(mock_server)
    .await;
```



## Tip 6:

Use asserts liberally



```
pub async fn cancel_build(
    client: &GitHubClient,
    db: &PgDbClient,
    build: &BuildModel,
) {
    assert_eq!(
        build.status,
        BuildStatus::Pending,
        "Passed a non-pending build to `cancel_build`"
    );
    ...
}
```



## Tip 7:

Leverage compile-time tests



```
fn merge_branch(repo: GitHubRepo, name: String)
    -> anyhow::Result<MergedBranch> {
    ...
}
```



Test

Test

Test

```
fn merge_branch(repo: GitHubRepo, name: String)
    -> anyhow::Result<MergedBranch> {
    ...
}
```



```
let build_id = sqlx::query_scalar!(
    r#"
INSERT INTO build (repository, branch, commit_sha, parent, state)
VALUES ($1, $2, $3, $4, $5)
RETURNING id
"#
    repo as &GithubRepoName,
    branch,
    commit_sha.0,
    parent.0,
    BuildStatus::Pending as BuildStatus
)
.fetch_one(executor)
.await?;
```



```
let build_id = sqlx::query_scalar!()
    error: error returned from database: column "state" of relation "build" does not exist
INS --> src/database/operations.rs:529:24
VAL |
RET 529 |     let build_id = sqlx::query_scalar!(
"#" , |     -----^
530 |     r#
531 |     INSERT INTO build (repository, branch, commit_sha, parent, state)
532 |     VALUES ($1, $2, $3, $4, $5)
...
539 |     BuildStatus::Pending as BuildStatus
)
540 |     )
.fe |     -----^
.await?;
```

```
pub struct Attribute {
    pub kind: AttrKind,
    pub id: AttrId,
    pub style: AttrStyle,
    pub span: Span,
}

const AssertAttributeSize: [(); 32] =
    [(); std::mem::size_of::<Attribute>()];
```



```
pub struct Attribute {
    pub kind: AttrKind,
    pub id: AttrId,
    pub style: AttrStyle,
    pub span: Span,
}

const AssertAttributeSize: [(); 32] =
    [(); std::mem::size_of::<Attribute>()];
```

```
error[E0308]: mismatched types
--> <anon>:15:5
|
14| const AssertAttributeSize: [(); 32] =
|     -- help: consider specifying the actual array length: `48`
15|     [(); std::mem::size_of::<Attribute>()]);
|     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ expected an array with a size of 32, found one with a size of 48
```



## Tip 8:

You can test (almost) anything



✓ **Kobzol** approved these changes on Mar 14 [View reviewed changes](#)

**Kobzol** left a comment

LGTM!



Member ...



In production:

```
ERROR:  column "base_branch" of relation "pull_request" contains null values
```



# Oops

```
✗ ...tions/20250313013946_add_base_branch_to_pr.up.sql ⌂
...
@@ -0,0 +1,2 @@
1 + -- Add up migration script here
2 + ALTER TABLE pull_request ADD COLUMN base_branch TEXT NOT NULL;
```



# Add a warning for next time:

## Updating the DB schema

### ⚠ Caution

When adding a new `NOT NULL` column, always specify the `DEFAULT` value that will be backfilled during the migration! Otherwise, the migration might break the deployed bors service.



# Add a warning for next time:

## Updating the DB schema

### ⚠ Caution

When adding a new `NOT NULL` column, always specify the `DEFAULT` value that will be backfilled during the migration! Otherwise, the migration might break the deployed bors service.

...is that all we can do?



```
▼ Cargo.toml ⌂ ⌄
.... @@ -63,6 +63,7 @@ tracing-test = "0.2.4"
63     63     regex = "1.10.4"
64     64     parking_lot = "0.12.3"
65     65     thread_local = "1.1.8"
66 + sqlparser = { version = "0.55", features = ["visitor"] }
66     67
```



```
#[test]
fn check_non_null_column_without_default() {
    let root = env!("CARGO_MANIFEST_DIR");
    let migrations = PathBuf::from(root).join("migrations");

}
```



```
#[test]
fn check_non_null_column_without_default() {
    let root = env!("CARGO_MANIFEST_DIR");
    let migrations = PathBuf::from(root).join("migrations");
    for file in std::fs::read_dir(migrations).unwrap() {
        let file = file.unwrap();
        if file.path().extension() == Some(OsStr::new("sql")) {
            let contents =
                std::fs::read_to_string(&file.path()).unwrap();
        }
    }
}
```



```
#[test]
fn check_non_null_column_without_default() {
    let root = env!("CARGO_MANIFEST_DIR");
    let migrations = PathBuf::from(root).join("migrations");
    for file in std::fs::read_dir(migrations).unwrap() {
        let file = file.unwrap();
        if file.path().extension() == Some(OsStr::new("sql")) {
            let contents =
                std::fs::read_to_string(&file.path()).unwrap();

            let ast = Parser::parse_sql(&PostgreSqlDialect {}, &contents).unwrap();
            let mut visitor = CheckNotNullWithoutDefault::new();
            ast.visit(&mut visitor);

            if let Some(error) = visitor.compute_error() {
                panic!(
                    "Migration {} contains error: {}",
                    file.path().display()
                );
            }
        }
    }
}
```

```
impl Visitor for CheckNotNullWithoutDefault {
    fn pre_visit_statement(&mut self, statement: &Statement)
        -> ControlFlow<Self::Break> {
        let Statement::AlterTable {
            operations, name, ..
        } = statement else {
            return ControlFlow::Continue(());
        }

        ControlFlow::Continue(())
    }
}
```



```
impl Visitor for CheckNotNullWithoutDefault {
    fn pre_visit_statement(&mut self, statement: &Statement)
        -> ControlFlow<Self::Break> {
        let Statement::AlterTable {
            operations, name, ..
        } = statement else {
            return ControlFlow::Continue(());
        };
        for op in operations {
            match op {
                AlterTableOperation::AlterColumn { column_name, op } => match op {
                    AlterColumnOperation::SetNotNull => {
                        self.columns_set_to_not_null
                            .insert((name.clone(), column_name.clone()));
                    }
                    AlterColumnOperation::SetDefault { .. } => {
                        self.columns_set_default_value
                            .insert((name.clone(), column_name.clone()));
                    }
                    _ => {}
                },
                _ => {}
            }
        }
        ControlFlow::Continue(())
    }
}
```





Didn't I just test implementation details?





Didn't I just test implementation details? 

Testing a *property* of an implementation



Didn't I just test implementation details? 

Testing a *property* of an implementation

To provide a good error message



## ✓ tests/data/migrations/20240518024921\_create\_pr.sql

```
... @@ -0,0 +1,7 @@
1 + INSERT INTO
2 +     pull_request (repository, number, build_id)
3 + VALUES
4 +     ('rust-lang/bors', 269, 1),
5 +     ('rust-lang/cargo', 14718, 2),
6 +     ('rust-lang/rust', 136864, 3),
7 +     ('rust-lang/clippy', 10521, NULL);
```



```
#[sqlx::test(migrations = false)]
async fn apply_migrations_with_test_data(pool: PgPool) -> anyhow::Result<()> {
    let migrations = get_sorted_up_migrations();

    for migration_path in migrations {

    }

    Ok(())
}
```



```
#[sqlx::test(migrations = false)]
async fn apply_migrations_with_test_data(pool: PgPool) -> anyhow::Result<()> {
    let migrations = get_sorted_up_migrations();

    for migration_path in migrations {
        let migration_sql = std::fs::read_to_string(&migration_path)?;
        pool.execute(&*migration_sql).await?;

        let test_data_path = get_test_data_path(&migration_path);
        let test_data = std::fs::read_to_string(&test_data_path)?;

        pool.execute(&*test_data).await.unwrap_or_else(|e| {
            panic!(
                "Failed to apply migration test data {:?}: {}",
                test_data_path, e
            )
        });
    }
    Ok(())
}
```



```
#[test]
fn check_migrations_have_sample_data() {
    let migrations = get_sorted_up_migrations();
    assert!(!migrations.is_empty());
    for migration_path in migrations {
        let test_data_path = get_test_data_path(&migration_path);

        assert!(
            test_data_path.exists(),
            "Migration {:?} does not have a test data file at {:?}.
Add a test data file there that fills some test data
into the database after that migration is applied.",
            migration_path,
            test_data_path
        );
    }
}
```



# cargo-wizard

```
> cargo wizard
? Select the profile that you want to update/create: █
> dev (builtin)
release (builtin)
<Create a new profile>
[↑↓ to move, enter to select, type to filter]
```



```
use reexpect::spawn;
use reexpect::error::Error;

fn main() -> Result<(), Error> {
    let mut p = spawn("cargo wizard", Some(2000))?;
    p.exp_regex(".*Select the profile.*")?;
    p.send("\x1b\x5b\x42")?; // Arrow down
    p.flush()?;
    p.send_line("")?;           // Enter
    Ok(())
}
```



```
#[test]
fn dialog_fast_compile_nightly() -> anyhow::Result<()> {
    let project = init_cargo_project()?;

    DialogBuilder::default()
        .nightly()
        .customize_item(
            "Number of frontend threads",
            CustomValue::Custom("4".to_string()),
        )
        .run(&project)?;

    insta::assert_snapshot!(project.read_config(), @r###"
[build]
rustflags = ["-Clink-arg=-fuse-ld=lld", "-Zthreads=4"]
"###);

    Ok(())
}
```



# Things that can be tests:



## Things that can be tests:

- Is all code properly formatted?



## Things that can be tests:

- Is all code properly formatted?
- Do all dependencies use compatible licenses?



## Things that can be tests:

- Is all code properly formatted?
- Do all dependencies use compatible licenses?
- Do we have a merge/unsigned commit in git history?



# Things that can be tests:

- Is all code properly formatted?
- Do all dependencies use compatible licenses?
- Do we have a merge/unsigned commit in git history?
- Can the example config file in the repo root be parsed?



## Tip 9:

Test (also) in production



# Problems with testing only in staging:



## Problems with testing only in staging:

- It's difficult to emulate production



## Problems with testing only in staging:

- It's difficult to emulate production
- It's difficult to get things right on the first try



Testing in production != testing on real users



# Feature flags

```
if is_feature_enabled("foo") {  
    do_a();  
} else {  
    do_b();  
}
```



## Annoyance 5:

Async ~~Rust~~ tests



```
async fn yolo() {  
    // Good luck testing this! xoxo  
    tokio::task::spawn(async move {  
        ...  
    }) ;  
}
```



## Tip 10:

Be in control of your async code



## Tip 10:

Be in control of your async code  
a.k.a. Use structured concurrency



## Tip 10:

Be in control of your async code

a.k.a. Use structured concurrency

a.k.a. Synchronize tests to avoid flaky race conditions



```
use tokio::time::interval;

async fn background_job() {
    let mut interval = interval(Duration::from_secs(30));
    loop {
        interval.tick().await;
        do_work().await;
    }
}
```



```
use tokio::time::interval;

async fn background_job() {
    // Duration might have to be adjusted for tests
    let mut interval = interval(Duration::from_secs(30));
    loop {
        interval.tick().await;
        do_work().await;
    }
}
```



```
use tokio::time::interval;

async fn background_job() {
    // Duration might have to be adjusted for tests
    let mut interval = interval(Duration::from_secs(30));
    loop { // The process runs forever
        interval.tick().await;
        do_work().await;
    }
}
```



```
use tokio::time::interval;

async fn background_job() {
    // Duration might have to be adjusted for tests
    let mut interval = interval(Duration::from_secs(30));
    loop { // The process runs forever
        interval.tick().await;
        do_work().await; // Not synchronized
    }
}
```



```
async fn background_job(mut rx: Receiver<()>) {  
    while let Some(msg) = rx.recv().await {  
        do_work().await  
    }  
}
```



```
async fn background_job(mut rx: Receiver<()>) {
    // Explicit control + proper cleanup
    while let Some(msg) = rx.recv().await {
        do_work().await
    }
}
```



```
async fn background_job(mut rx: Receiver<()>) {
    // Explicit control + proper cleanup
    while let Some(msg) = rx.recv().await {
        do_work().await
    }
}

async fn driver(tx: Sender<()>) {
    let mut interval = interval(Duration::from_secs(30));
    loop {
        interval.tick().await;
        tx.send(()).await;
    }
}
```



```
async fn background_job(mut rx: Receiver<()>) {
    // Explicit control + proper cleanup
    while let Some(msg) = rx.recv().await {
        do_work().await
    }
}

async fn test() {
    let (tx, rx) = mpsc::channel(10);
    task::spawn(background_job(rx));

    do_something().await;
    tx.send(()).await;
    ensure_work_was_done().await;
}
```



```
async fn background_job(mut rx: Receiver<()>) {
    // Explicit control + proper cleanup
    while let Some(msg) = rx.recv().await {
        do_work().await
    }
}

async fn test() {
    let (tx, rx) = mpsc::channel(10);
    task::spawn(background_job(rx));

    do_something().await;
    tx.send(()).await;
    ensure_work_was_done().await; ↪ Possible race!
}
```



# How to ensure that something *happens* reliably?



# How to ensure that something *happens* reliably?

- Arbitrarily add sleeps throughout tests until it "works" 



# How to ensure that something *happens* reliably?

- Arbitrarily add sleeps throughout tests until it "works" 
- Synchronize through some external state
  - Database, HTTP endpoint, file on disk, ...



```
async fn unapprove() {  
    ...  
    bors.post_comment("@bors r-") .await?  
  
    bors.get_pr() .await .expect_unapproved();  
}
```



```
async fn unapprove() {
    ...
    bors.post_comment("@bors r-").await?;
    // Bot comment acts as a synchronization point
    insta::assert_snapshot!(
        bors.get_next_comment().await?,
        @"Commit pr-1-sha has been unapproved."
    );
    bors.get_pr().await.expect_unapproved();
}
```



# How to ensure that something *happens* reliably?

- Arbitrarily add sleeps throughout tests until it "works" 
- Synchronize through some external state
  - Database, HTTP endpoint, file on disk, ...
- Use coverage marks



```
#[cfg(test)]  
pub static WAIT_FOR_WORKFLOW_STARTED = TestSyncMarker::new();
```



```
#[cfg(test)]
pub static WAIT_FOR_WORKFLOW_STARTED = TestSyncMarker::new();

fn insides_of_bors() {
    ...
    BorsRepositoryEvent::WorkflowStarted(payload) => {
        handle_workflow_started(repo, db, payload).await?;
    }
    ...
}
```



```
#[cfg(test)]
pub static WAIT_FOR_WORKFLOW_STARTED = TestSyncMarker::new();

fn insides_of_bors() {
    ...
    BorsRepositoryEvent::WorkflowStarted(payload) => {
        handle_workflow_started(repo, db, payload).await?;
    }
    ...
}

#[cfg(test)]
async fn test() {
    bors.start_workflow().await;
    WAIT_FOR_WORKFLOW_STARTED.sync().await;
    bors.assert_workflow_started().await;
}
```



## Tip 11:

Collect all errors during tests



```
async fn test_approve() -> anyhow::Result<()> {
    let bors = start_bors().await;
    bors.post_comment("@bors r+").await?;
    ...
    Ok(())
}
```



```
async fn test_approve() -> anyhow::Result<()> {
    let bors = start_bors().await;
    bors.post_comment("@bors r+").await?;
    ...
    Ok(())
}
```

Error: Connection reset by peer (os error 104)



```
async fn test_approve() -> anyhow::Result<()> {
    let bors = start_bors().await;
    bors.post_comment("@bors r+").await?;
    ...
    Ok(())
}
```

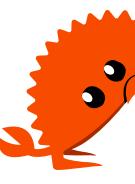
Error: Connection reset by peer (os error 104)

thread '<unnamed>' (269367) panicked at src/lib.rs:229:25:  
Cannot approve PR 1 because of ...



```
async fn run_test<F>(test_fn: F) -> anyhow::Result<()>
where
    F: AsyncFnOnce(&mut Ctx) -> anyhow::Result<()>
{
```

}



```
async fn run_test<F>(test_fn: F) -> anyhow::Result<()>
where
    F: AsyncFnOnce(&mut Ctx) -> anyhow::Result<()>
{
    let mut ctx = create_ctx().await;

    // or test_fn(&mut ctx).catch_unwind()
    let res1 = test_fn(&mut ctx).await;
    let res2 = ctx.finish().await;
    combine_results(res1, res2)
}
```



```
async fn run_test<F>(test_fn: F) -> anyhow::Result<()>
where
    F: AsyncFnOnce(&mut Ctx) -> anyhow::Result<()>
{
    let mut ctx = create_ctx().await;

    // or test_fn(&mut ctx).catch_unwind()
    let res1 = test_fn(&mut ctx).await;
    let res2 = ctx.finish().await;
    combine_results(res1, res2)
}

#[tokio::test]
async fn test() {
    run_test(async |bors| {
        bors.post_comment("@bors r+").await?;
        Ok(())
    }).await;
}
```



## Some further inspiration:

- tokio `turmoil`



## Some further inspiration:

- tokio `turmoil`
- Deterministic Simulation Testing
- TigerBeetle
- antithesis



```
#[sqlx::test]
async fn unapprove_lacking_permissions(pool: sqlx::PgPool) {
    run_test(pool, async |bors| {
        bors.approve(()).await?;
        bors.post_comment(Comment::from("@bors r-")
            .with_author(User::unprivileged()))
        .await?;
        insta::assert_snapshot!(
            bors.get_next_comment_text().await?,
            "@@unprivileged-user: :key:
Insufficient privileges: not in review users"
        );
    });
}
```



## Annoyance 6:

Test metrics



## Overheard on the internet:

“

We only have 90% test coverage,  
but we want to reach 100% someday.

”



# Coverage types



# Coverage types

- Line coverage



# Coverage types

- Line coverage
  - This is usually what you get from tooling!



# Coverage types

- Line coverage
  - This is usually what you get from tooling!
- Branch coverage



# Coverage types

- Line coverage
  - This is usually what you get from tooling!
- Branch coverage
- Path coverage



# Coverage types

- Line coverage
  - This is usually what you get from tooling!
- Branch coverage
- Path coverage
  - Exponential!



```
fn get_host_without_tld(mut text: &str) -> &str {
    if let Some(pos) = text.find("@") {
        text = &text[pos..];
    }
    &text[text.find(".").unwrap()..]
}
```



```
fn get_host_without_tld(mut text: &str) -> &str {
    if let Some(pos) = text.find("@") {
        text = &text[pos..];
    }
    &text[text.find(".").unwrap()..]
}

#[test]
fn test1() {
    get_host_without_tld("foo@bar.cz"); // 100% line coverage
}
```



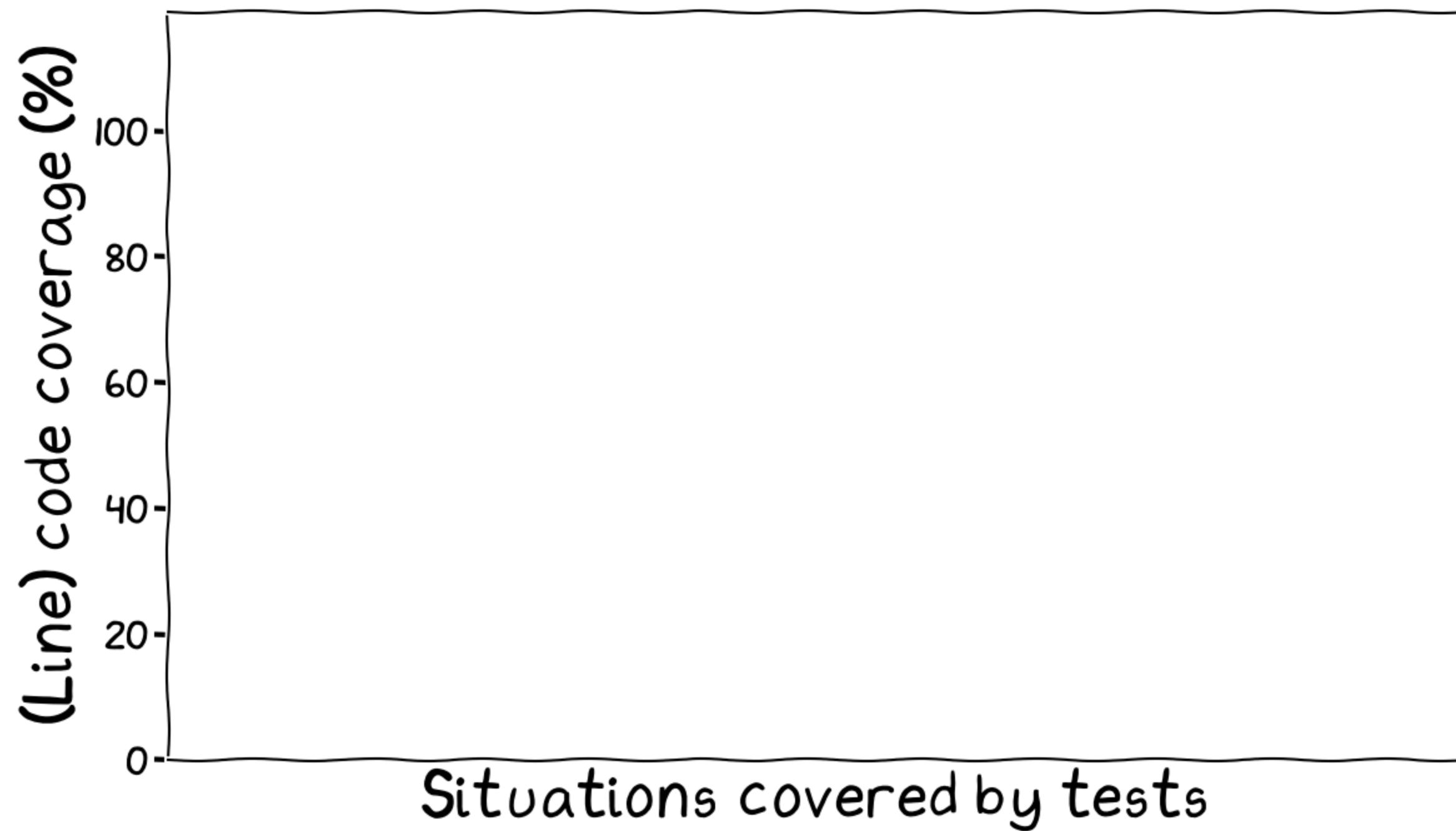
```
fn get_host_without_tld(mut text: &str) -> &str {
    if let Some(pos) = text.find("@") {
        text = &text[pos..];
    }
    &text[text.find(".").unwrap()..]
}

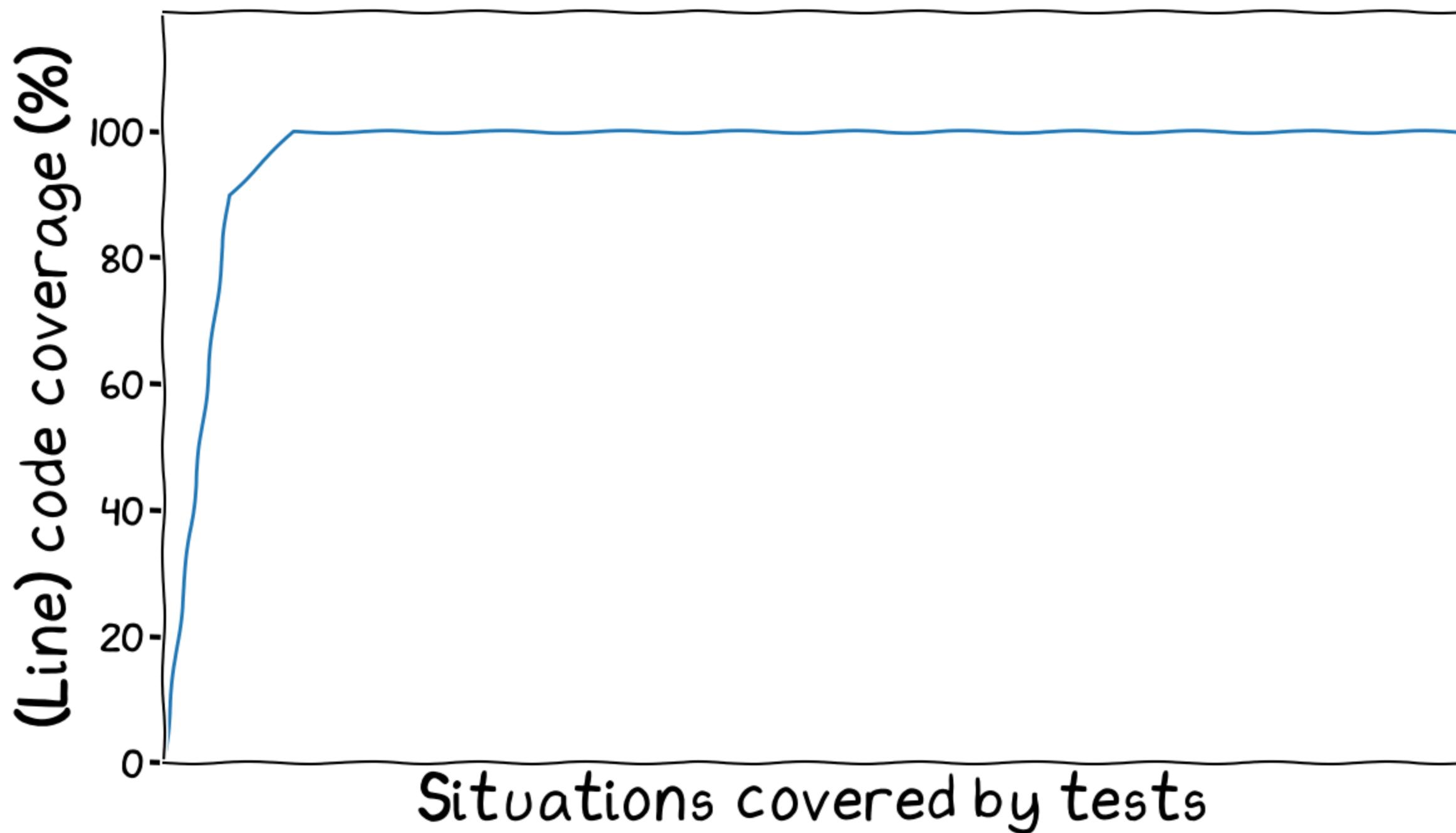
#[test]
fn test1() {
    get_host_without_tld("foo@bar.cz"); // 100% line coverage
    get_host_without_tld("bar.cz");     // 100% branch coverage
}
```



```
fn get_host_without_tld(mut text: &str) -> &str {
    if let Some(pos) = text.find("@") {
        text = &text[pos..];
    }
    &text[text.find(".").unwrap()..]
}

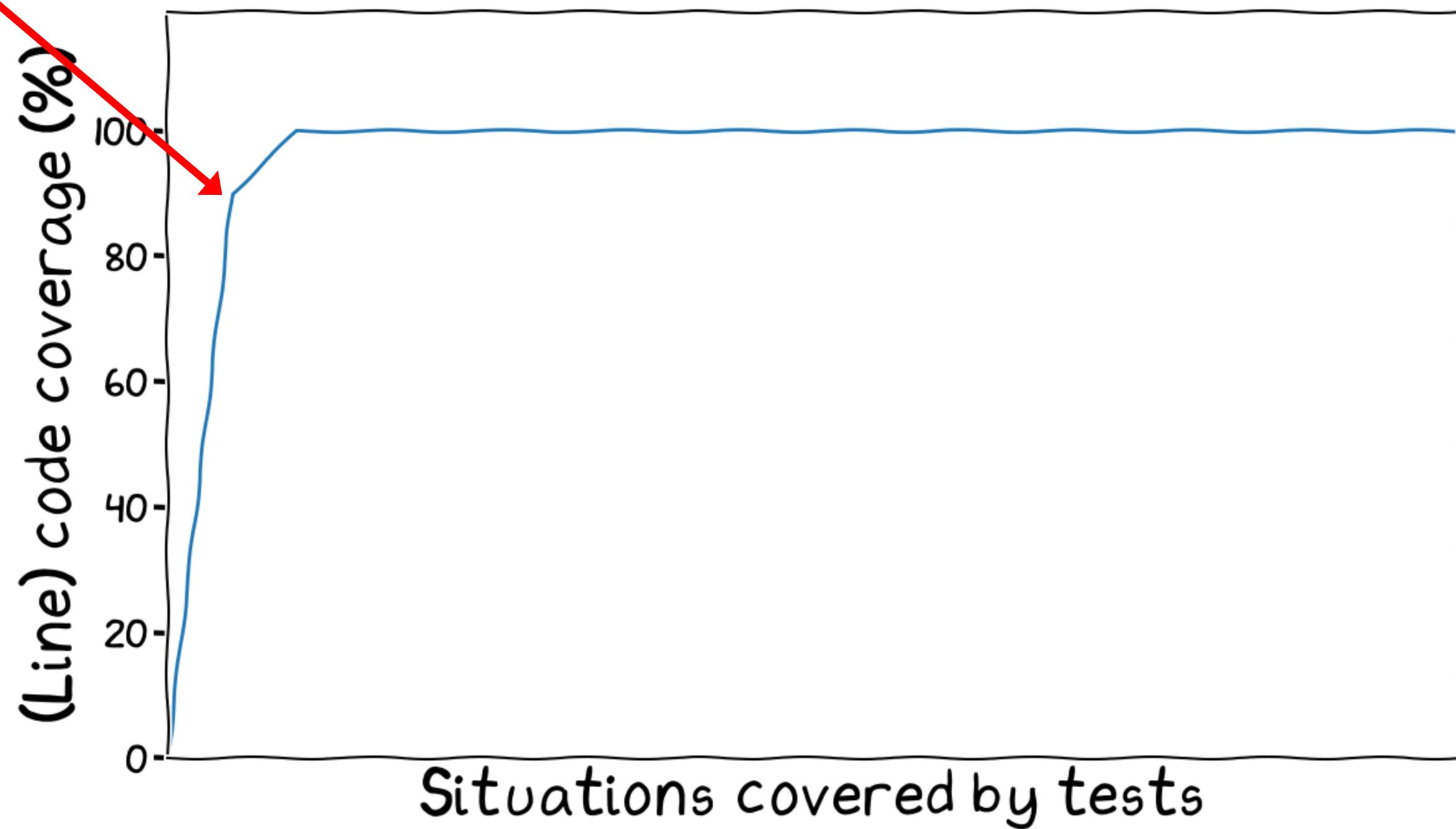
#[test]
fn test1() {
    get_host_without_tld("foo@bar.cz"); // 100% line coverage
    get_host_without_tld("bar.cz");      // 100% branch coverage
    get_host_without_tld("bar");         // Panic :-(
```

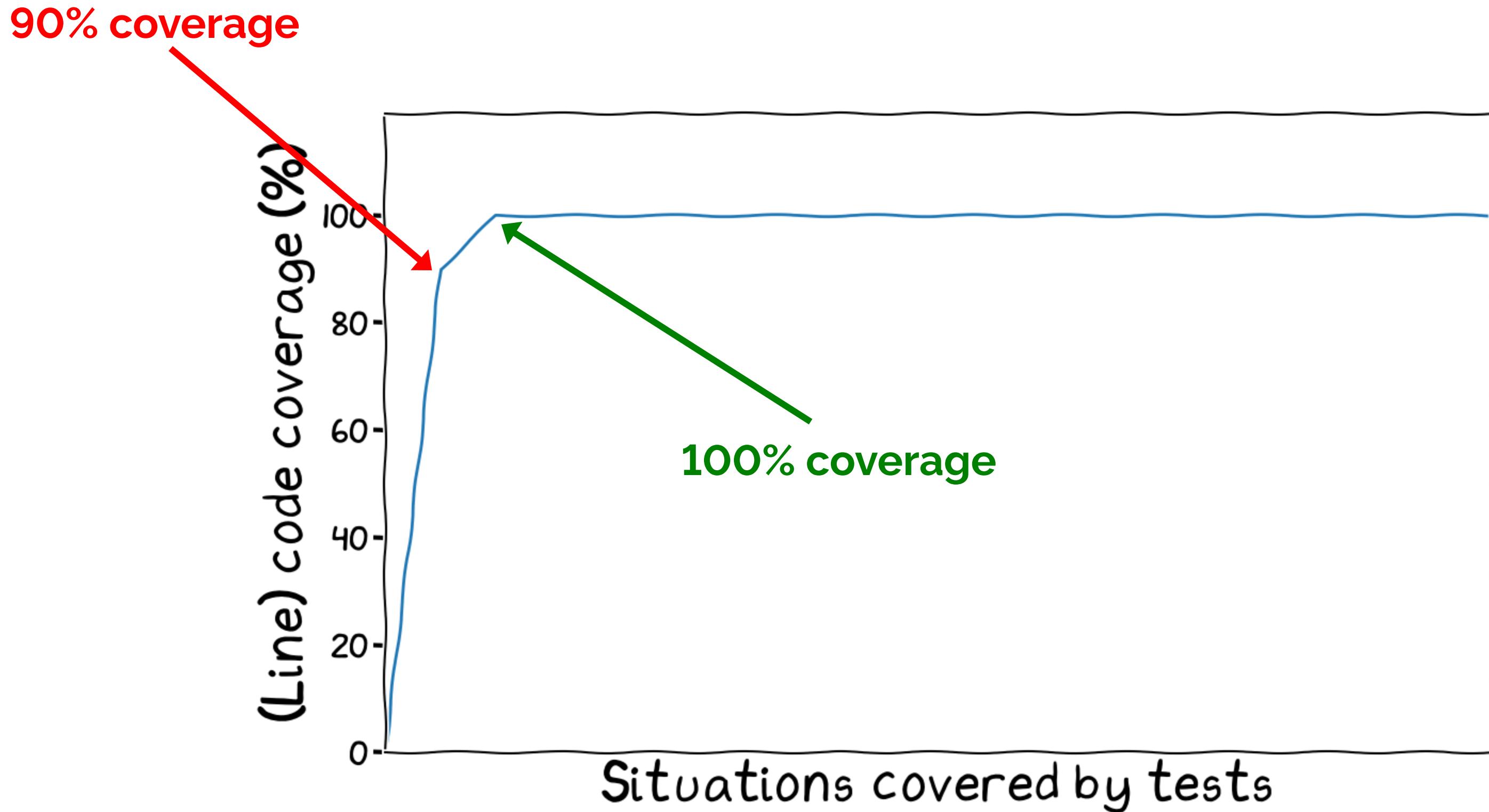






90% coverage

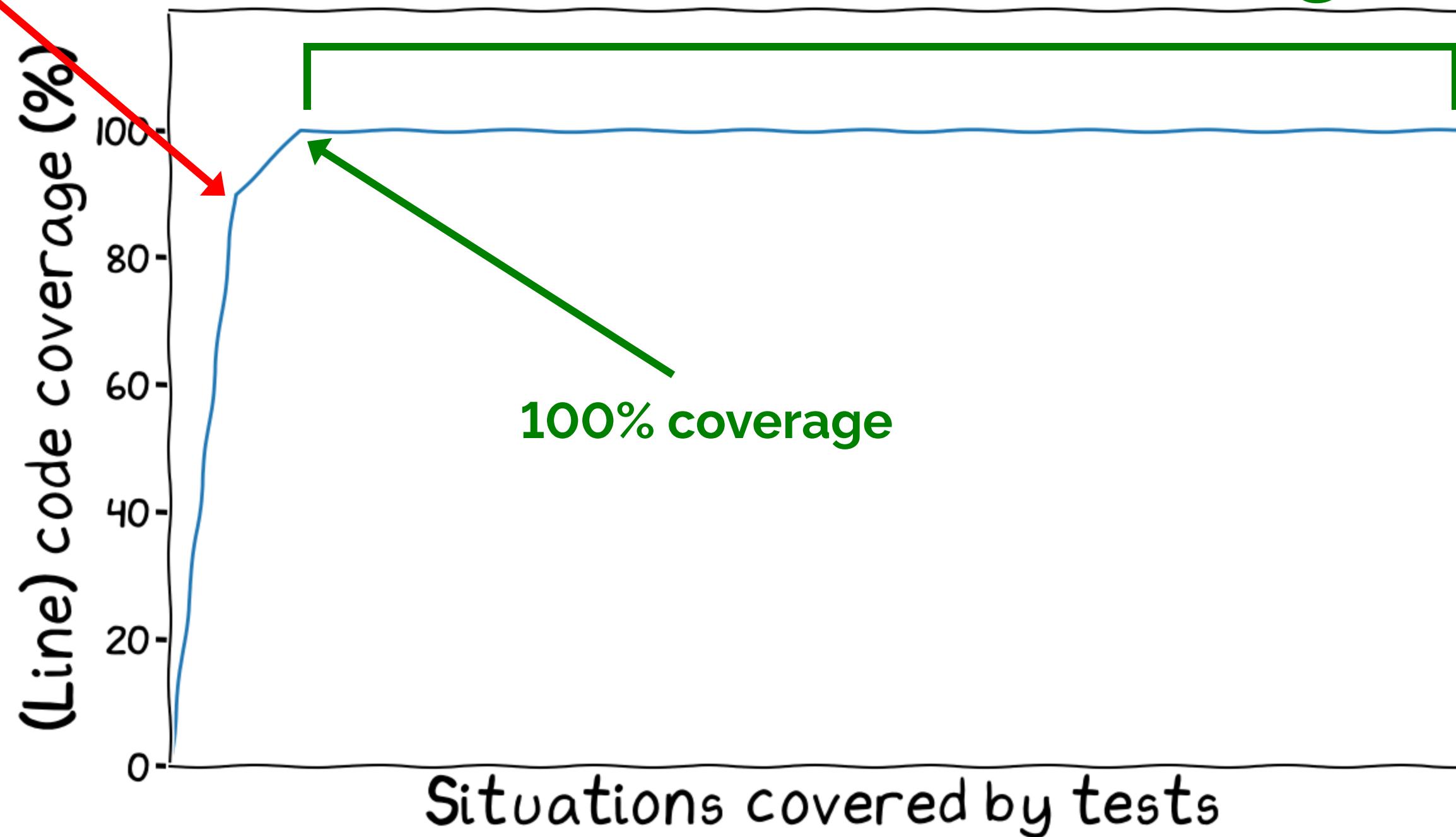






90% coverage

Also 100% coverage!





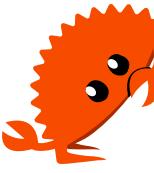
## Tip 12:

Metrics can be useful  
But don't worry about them too much



## Annoyance 7:

CI is green in PR, but red after merge

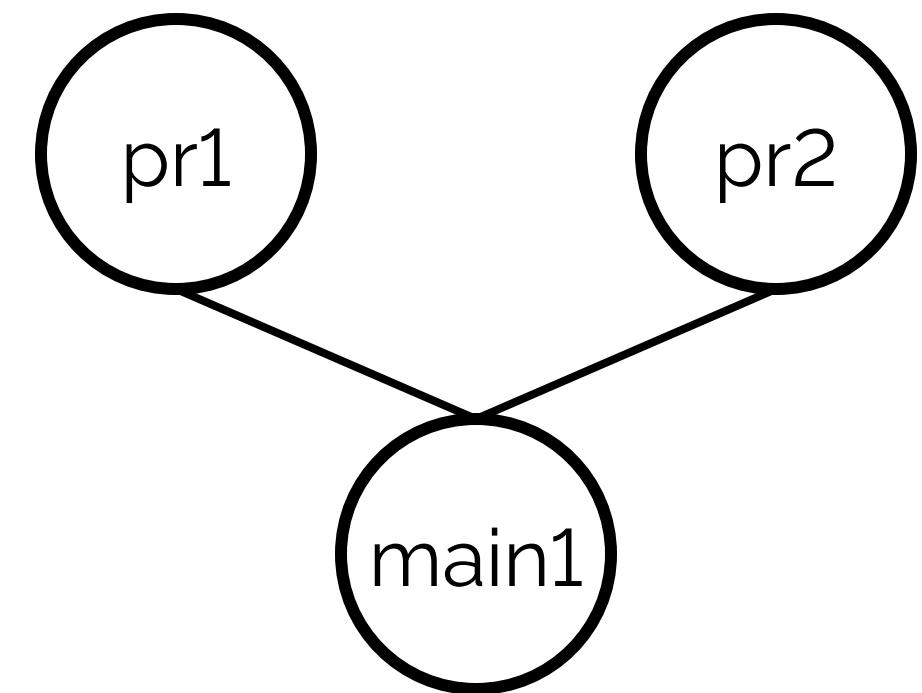
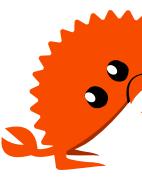


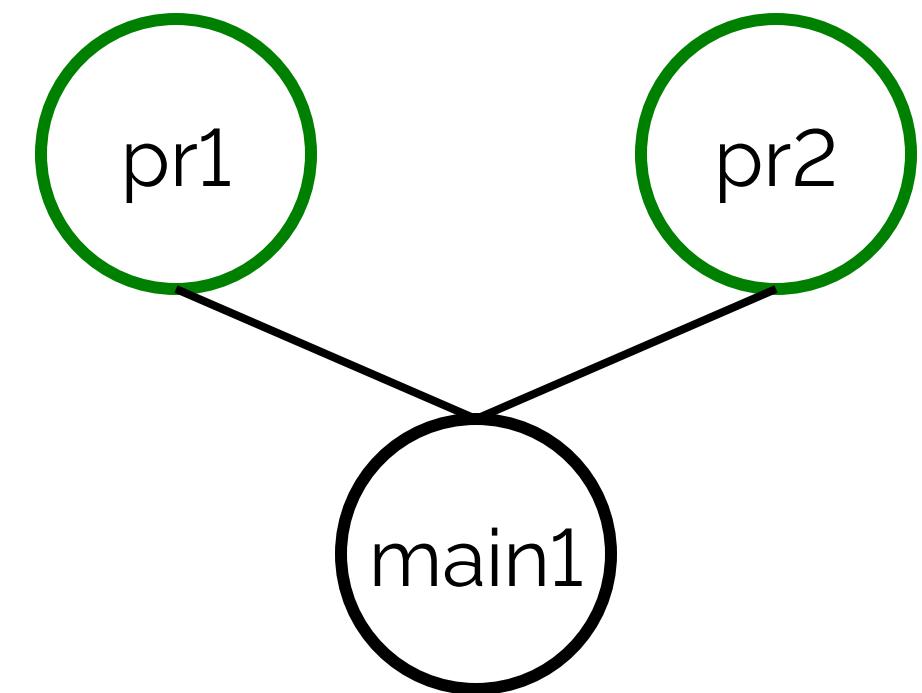
“

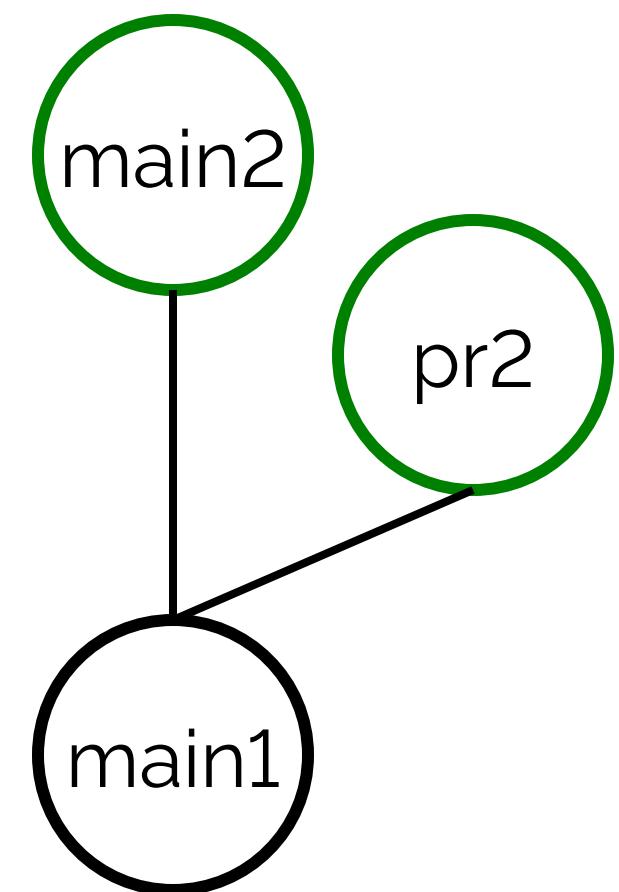
## **The Not Rocket Science Rule Of Software Engineering:**

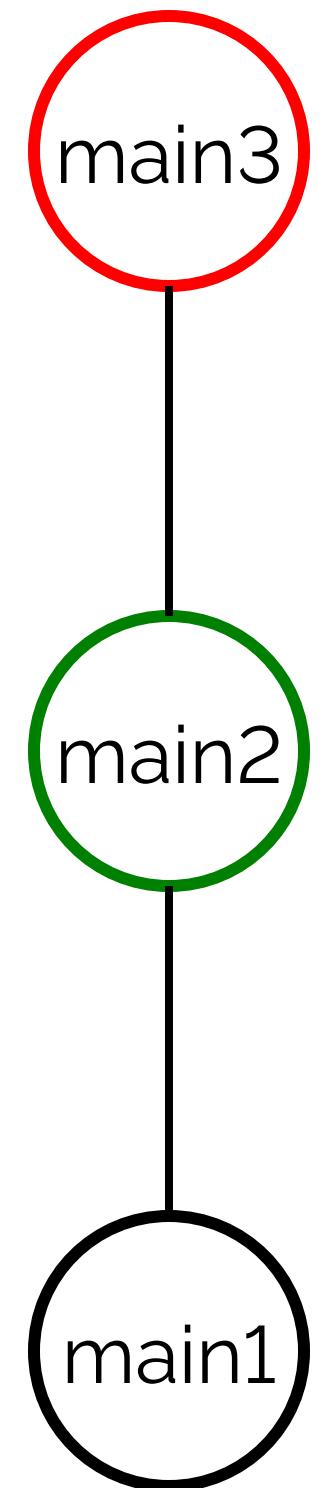
Automatically maintain a repository of code that always passes all the tests.

Graydon Hoare (creator of Rust) ,”











## Tip 13:

Use merge queues/trains



## Tip 14:

Invest in test infrastructure



## Tip 14:

Invest in test infrastructure

It is ~~100~~ worth it



## Tip 15:

Things will still fail somehow



## Tip 15:

Things will still fail somehow  
...so make peace with it



**Kobzol** commented 3 hours ago

Member

Author

...

@rust-timer queue





**Kobzol** commented 3 hours ago

Member

Author

...

@rust-timer queue



**Kobzol** commented 3 hours ago

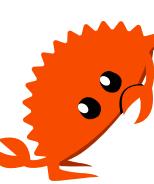
Member

Author

...

@rust-timer





```
153     let mut valid_build_cmds = vec![];
154     let mut errors = String::new();
155     for cmd in parse_build_commands(&comment.body) {
156         match cmd {
157             Ok(cmd) => valid_build_cmds.push(cmd),
158             Err(error) => errors.push_str(&format!("Cannot parse build command: {}{}\n"), error),
159         }
160     }
161 +
162 +     if valid_build_cmds.is_empty() {
163 +         errors.push_str("@rust-timer command cannot be empty\n");
164 +     }
165 +
166         if !errors.is_empty() {
167             main_client.post_comment(issue.number, errors).await;
```



 **rust-timer** commented 4 hours ago

@rust-timer command cannot be empty



Member ...



**rust-timer** commented 4 hours ago

Member

...

@rust-timer command cannot be empty



**rust-timer** commented 4 hours ago

Member

...

@rust-timer command cannot be empty



**rust-timer** commented 4 hours ago

Member

...

@rust-timer command cannot be empty





-  **rust-timer** commented 4 hours ago Member ...

@rust-timer command cannot be empty


-  **rust-timer** commented 4 hours ago Member ...

@rust-timer command cannot be empty


-  **rust-timer** commented 4 hours ago Member ...

@rust-timer command cannot be empty


-  **rust-timer** commented 4 hours ago Member ...

@rust-timer command cannot be empty


-  **rust-timer** commented 4 hours ago Member ...

@rust-timer command cannot be empty


-  **rust-timer** commented 4 hours ago Member ...

@rust-timer command cannot be empty

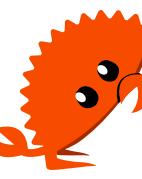

-  **rust-timer** commented 4 hours ago Member ...

@rust-timer command cannot be empty


-  **rust-timer** commented 4 hours ago Member ...

@rust-timer command cannot be empty





**565 hidden items**  
[\*\*Load more...\*\*](#)



## **TLDR:**

- Build high-level test APIs



## **TLDR:**

- Build high-level test APIs
- Use data-driven integration tests that are blesstable



## **TLDR:**

- Build high-level test APIs
- Use data-driven integration tests that are blesstable
- Make tests visual to ease understanding



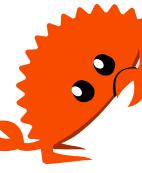
## **TLDR:**

- Build high-level test APIs
- Use data-driven integration tests that are blessable
- Make tests visual to ease understanding
- Test everything and anything



## **TLDR:**

- Build high-level test APIs
- Use data-driven integration tests that are blessable
- Make tests visual to ease understanding
- Test everything and anything
- Don't test nor mock (too many) implementation details



# Matklad's blog:

**<https://matklad.github.io/2021/05/31/how-to-test.html>**





**rustlang . cz**





# Thank you for your attention!

Slides are available here:



Slides were programmed using [github.com/spirali/elsie](https://github.com/spirali/elsie)  
(2328 lines of Python)

Several emojis were used from the Noto Emoji pack