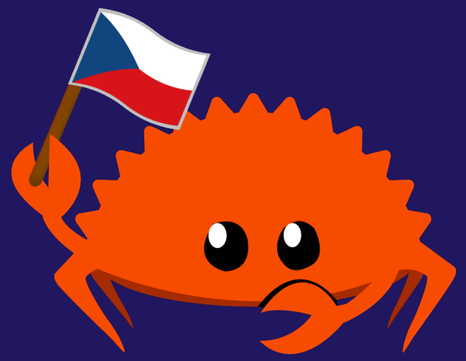
meetup



Tipy, jak psát lepší (Rust) testy

Kuba Beránek

- Lecturer @ VSB-TUO
- Researcher @ IT4Innovations
- Open-source contributor @ Rust Project

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

James Munns

GitHub: jamesmunns

Launching pad

Mara Bos

GitHub: m-ou-se

Library team

TC

GitHub: traviscross

Language team

Eric Huss

GitHub: ehuss

Dev tools team

Jakub Beránek

GitHub: Kobzol

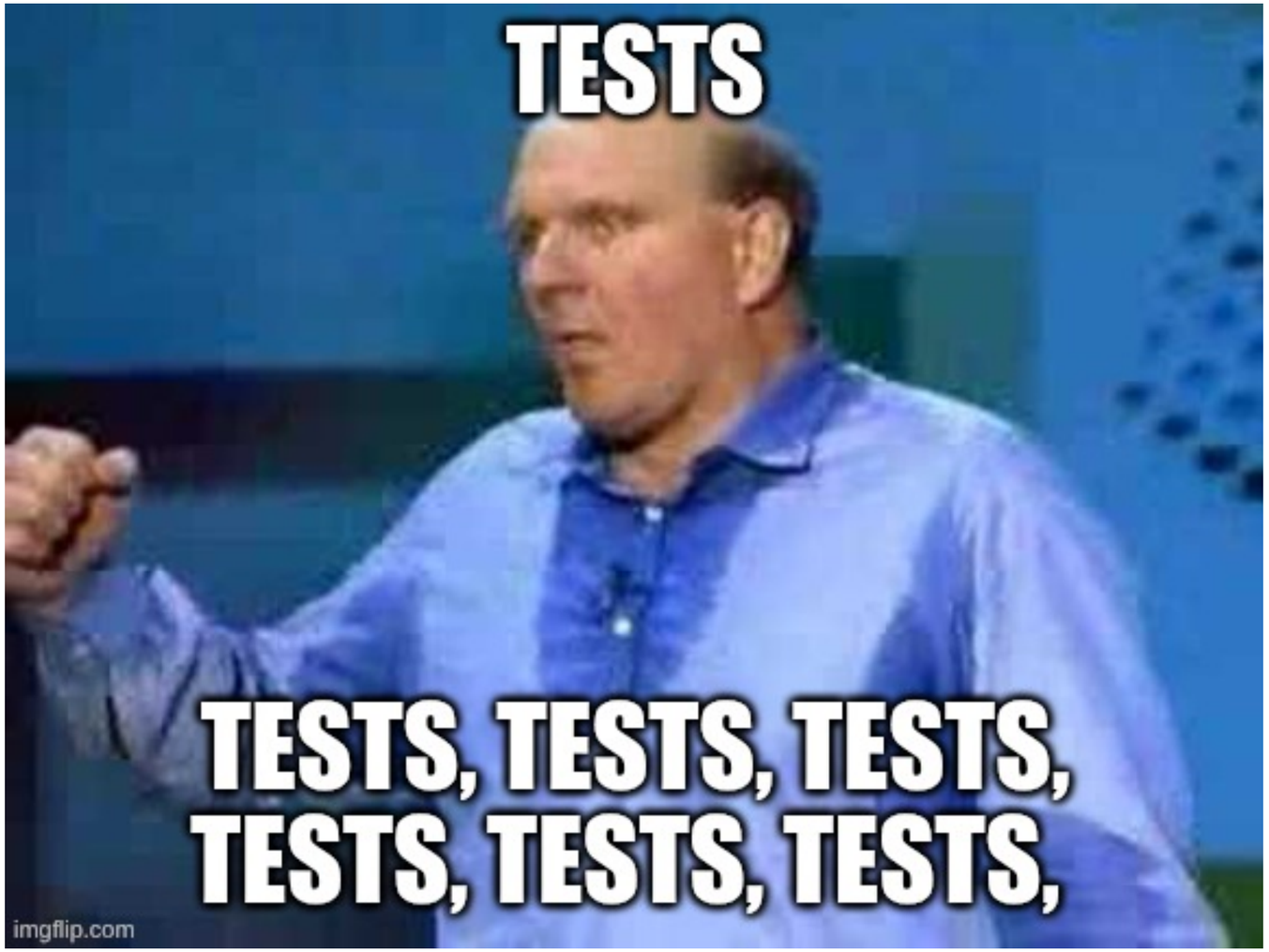
Infrastructure team

Oliver Scherer

GitHub: oli-obk

Moderation team

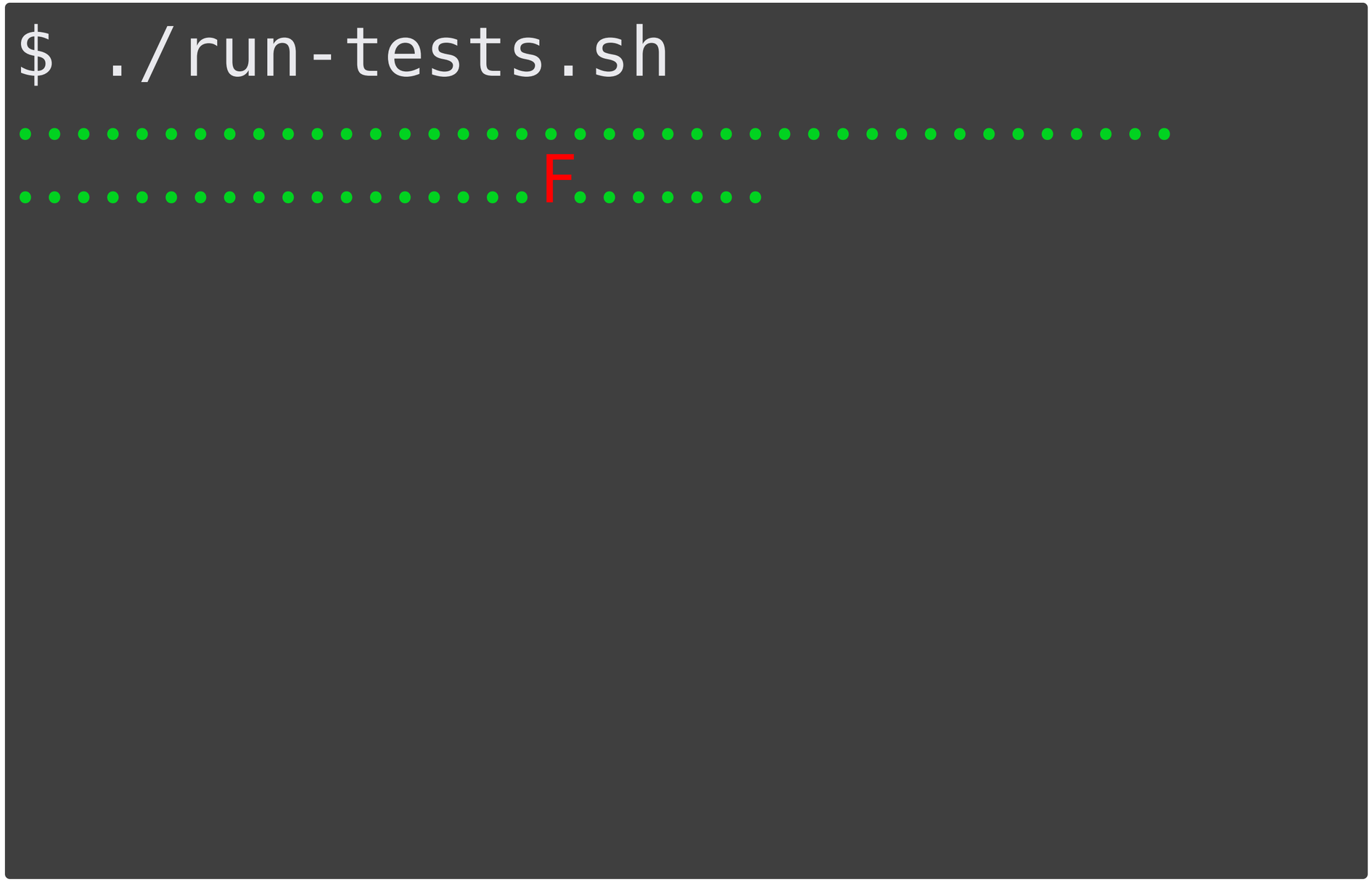
rustlang.cz

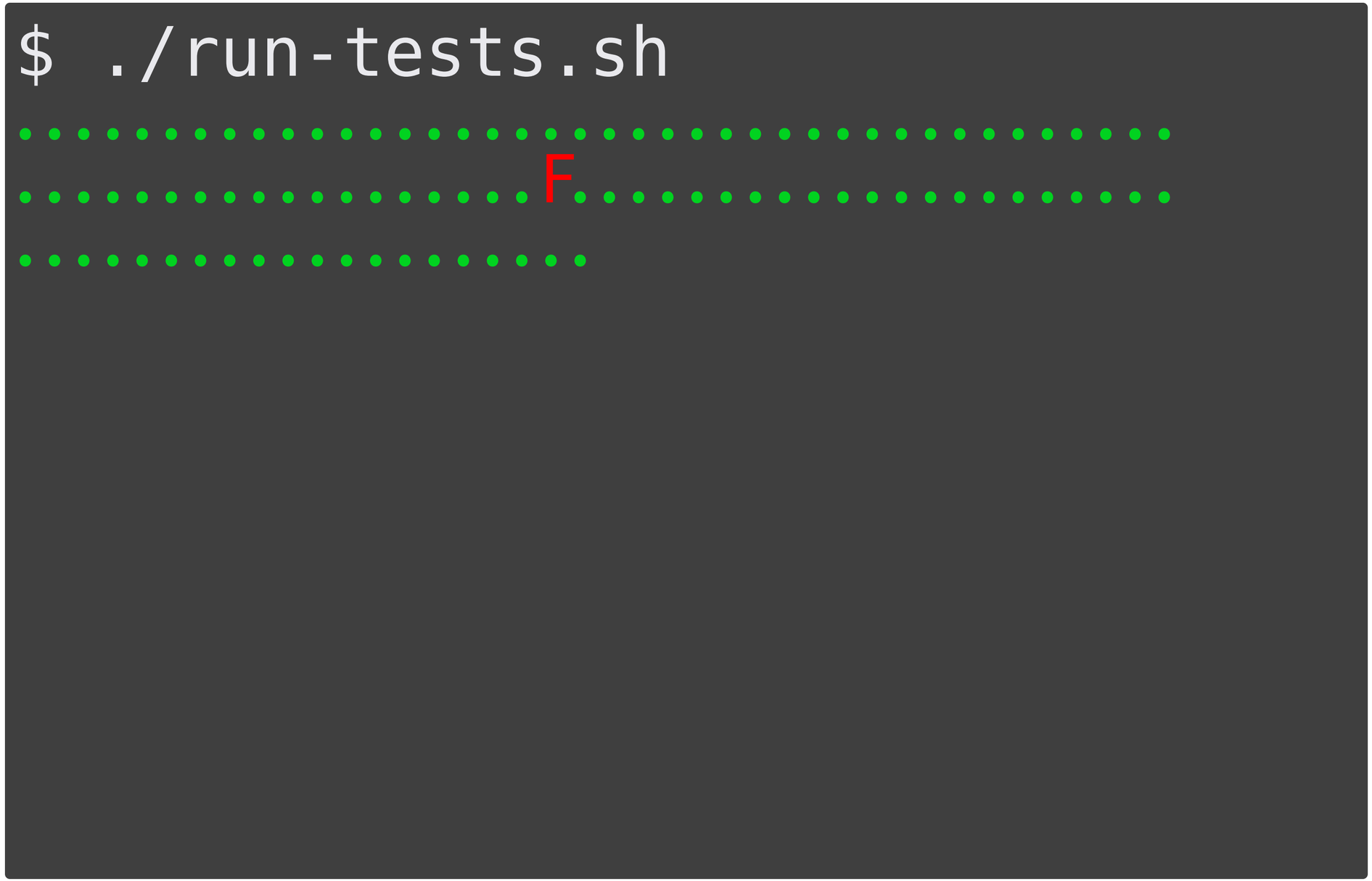

TESTS
TESTS, TESTS, TESTS,
TESTS, TESTS, TESTS,
imgflip.com

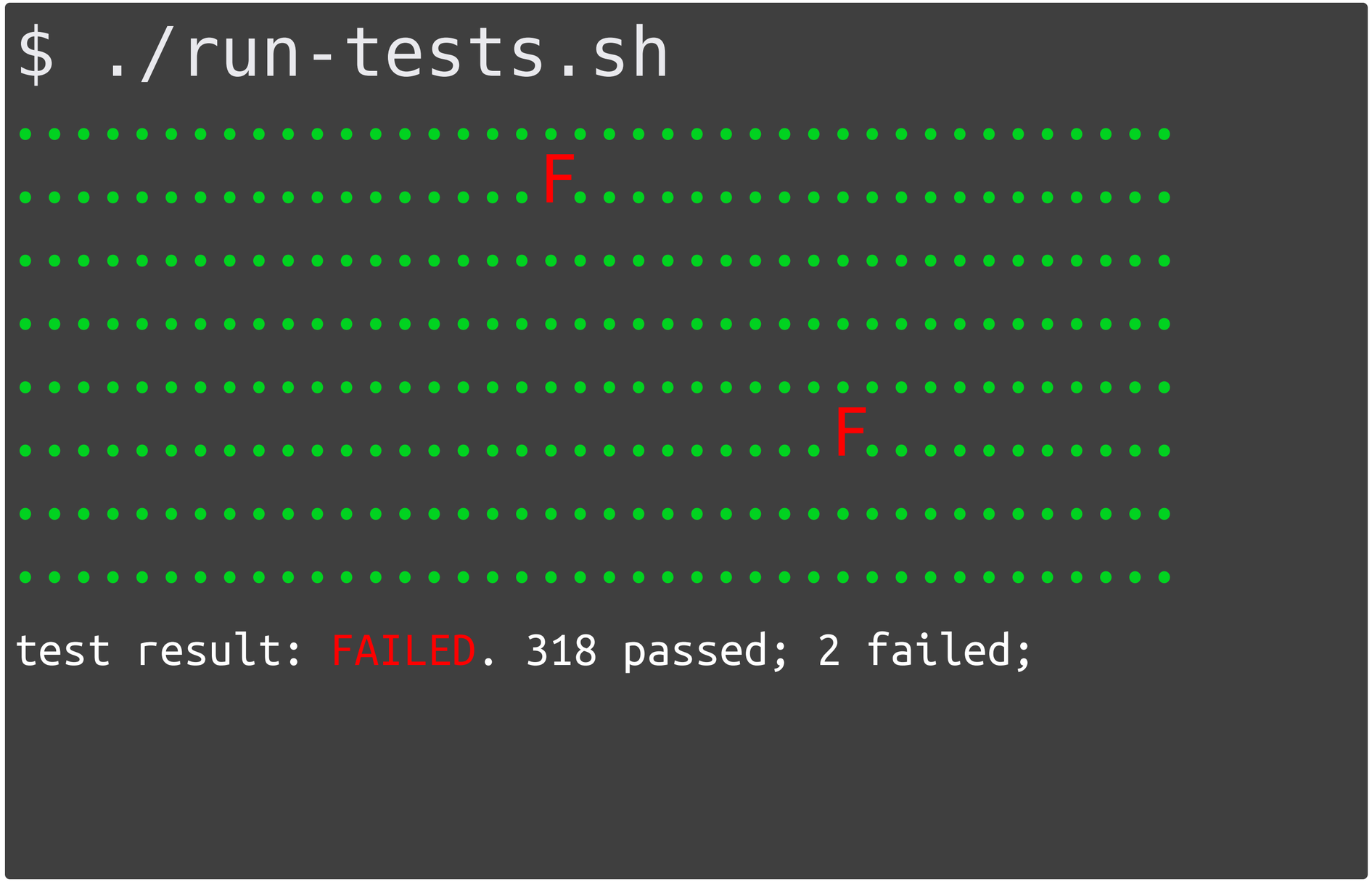
```
$ ./run-tests.sh
```

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

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


$ ./run-tests.sh
.....F.....


$ ./run-tests.sh
F

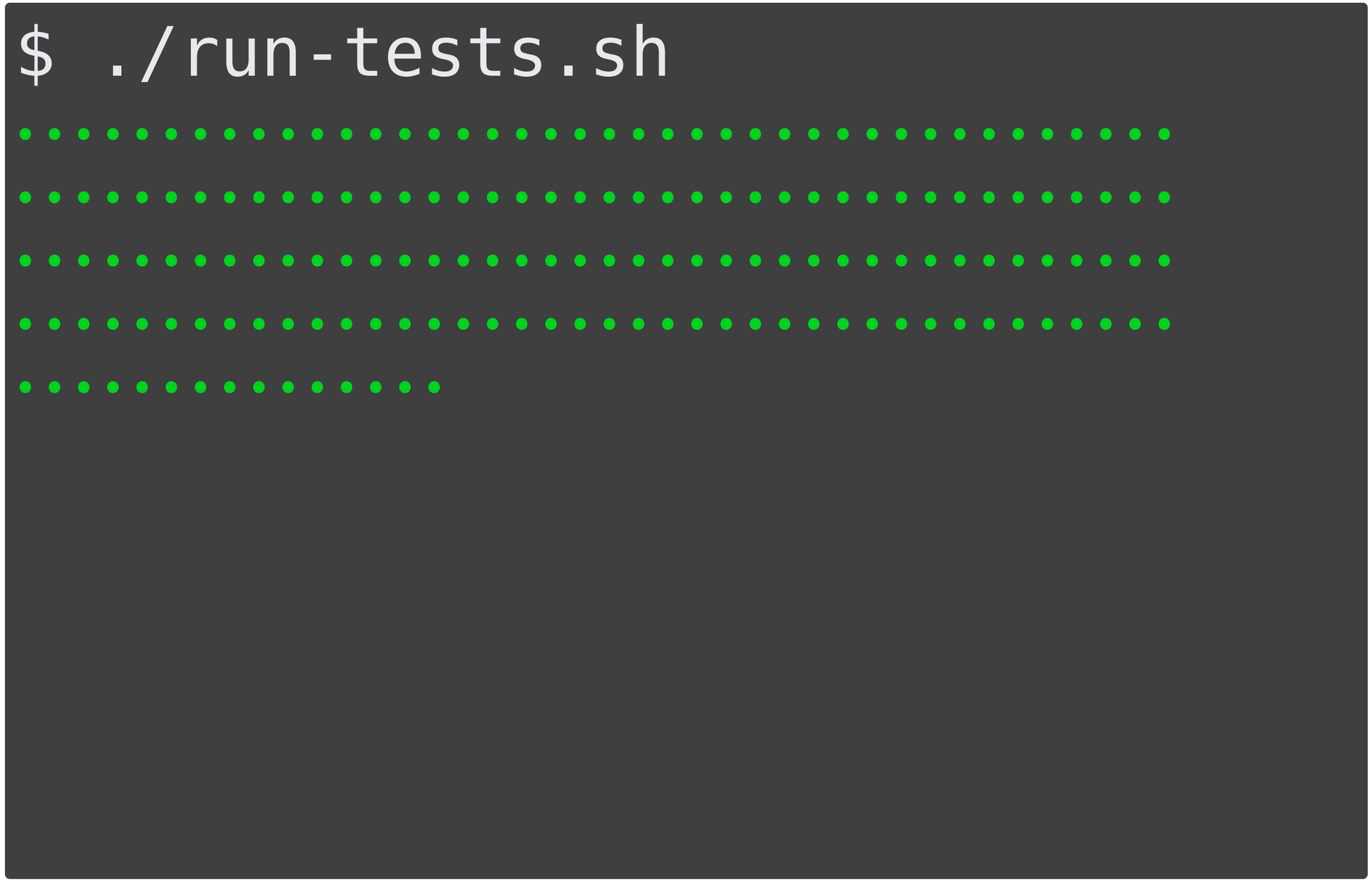

$ ./run-tests.sh
test result: FAILED. 318 passed; 2 failed;

```
$ ./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

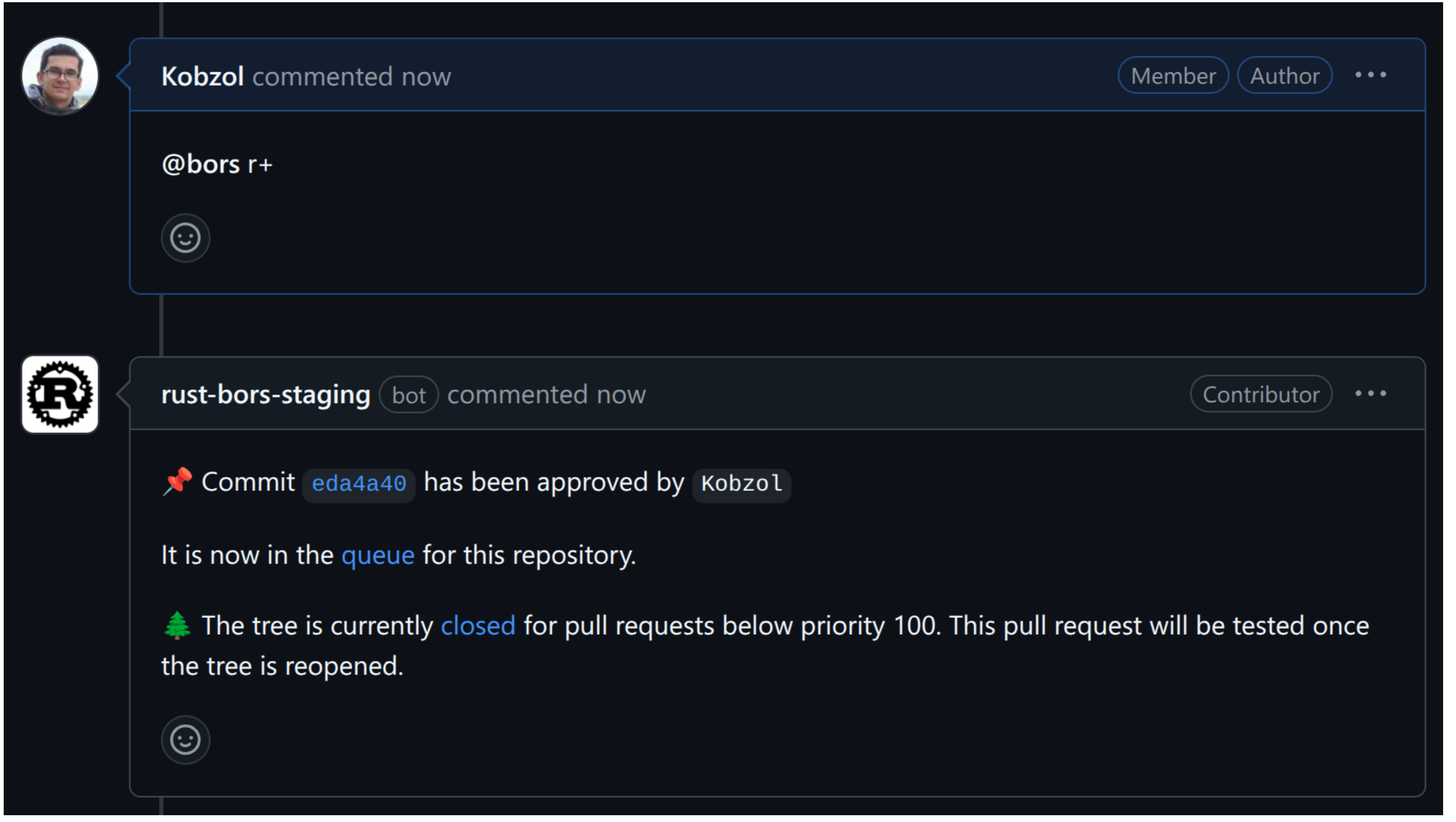
Tests kinda suck 😔

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

Can we make tests less annoying?

rust-lang/bors

GitHub merge queue bot


Kobzol commented now
Member
Author
@bors r+
rust-bors-staging bot commented now
Contributor
📌 Commit eda4a40 has been approved by Kobzol
It is now in the queue for this repository.
🌲 The tree is currently closed for pull requests below priority 100. This pull request will be tested once the tree is reopened.

Bors queue - rust

Create rollup 978 total, 13 in queue, 1 failed | Estimated time to flush queue: 1d 19h

Search: Group by: None

	#	Status	Mergeable	Title	Author
	151550	pending (30m)	yes	resolve: Replace `Macros20NormalizedIdent` with `IdentKey`	petrochenkov
☐	150491	approved	yes	resolve: Mark items under exported ambiguous imports as exported	petrochenkov
☐	150720	approved	yes	Do not suggest `derive` if there is already an impl	WhyNovaa
☐	150968	approved	yes	compiler-builtins: Remove the no-f16-f128 feature	tgross35
☐	151493	approved	yes	[RFC] rustc_parse: improve the error diagnostic for "missing let in let chain"	Unique-Usman

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
```

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

  repo.record_pr(pr);
  repo.set_pr_priority(5, 10);

  assert_eq!(repo.get_pr(5).get_priority(), 10);
}
```

Tip 1:

Build high-level test APIs

bors

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

    #[builder(default = 1)]
    number: u32,
    …
}
```

bors

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

    #[builder(default = 1)]
    number: u32,
    …
}
```

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

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

bors

```
let repo = Repository::new(...);
repo.approve_pr(1, User::new("karel"));
```

bors

```
let repo = Repository::new(...);
repo.approve_pr(1, User::new("karel"));

ctx.approve(1, "karel");
```

bors

```
repo.approve_pr(1, User::new("karel"));
assert!(repo.get_pr(1).get_approval().is_approved());
assert_eq!(repo.get_pr(1).get_approval().approver(), "karel");
assert_eq!(repo.get_pr(1).get_priority(), 0);
```

bors

```
repo.approve_pr(1, User::new("karel"));
assert!(repo.get_pr(1).get_approval().is_approved());
assert_eq!(repo.get_pr(1).get_approval().approver(), "karel");
assert_eq!(repo.get_pr(1).get_priority(), 0);

ctx.approve(1, "karel");
ctx.get_pr(1)
    .expect_approved_by("karel")
    .expect_priority(0);
```

bors

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

bors

```rust
#[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:
```

```
ctx.approve("foo/bar", 1, "karel");
ctx.approve("foo/bar", 1, "martin");
ctx.approve("foo/bar", 1, "kobzol");
ctx.approve("foo/bar", 2, "karel");
```

```
ctx.approve("foo/bar", 1, "karel");
ctx.approve("foo/bar", 1, "martin");
ctx.approve("foo/bar", 1, "kobzol");
ctx.approve("foo/bar", 2, "karel");

ctx.approve("karel");
ctx.approve_complex("foo/bar", 2, "karel");
```

```
ctx.approve("foo/bar", 1, "karel");
ctx.approve("foo/bar", 1, "martin");
ctx.approve("foo/bar", 1, "kobzol");
ctx.approve("foo/bar", 2, "karel");

ctx.approve("karel");
ctx.approve_complex("foo/bar", 2, "karel");

ctx.approve(Approval::new("karel").pr(2));
```

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

// The "Into trick":
fn approve<Id: Into<PrIdentifier>>(
    &mut self,
    id: Id,
    reviewer: &str
) { … }
```

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

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

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

```rust
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 }
    }
}
```

```rust
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 }
    }
}
```

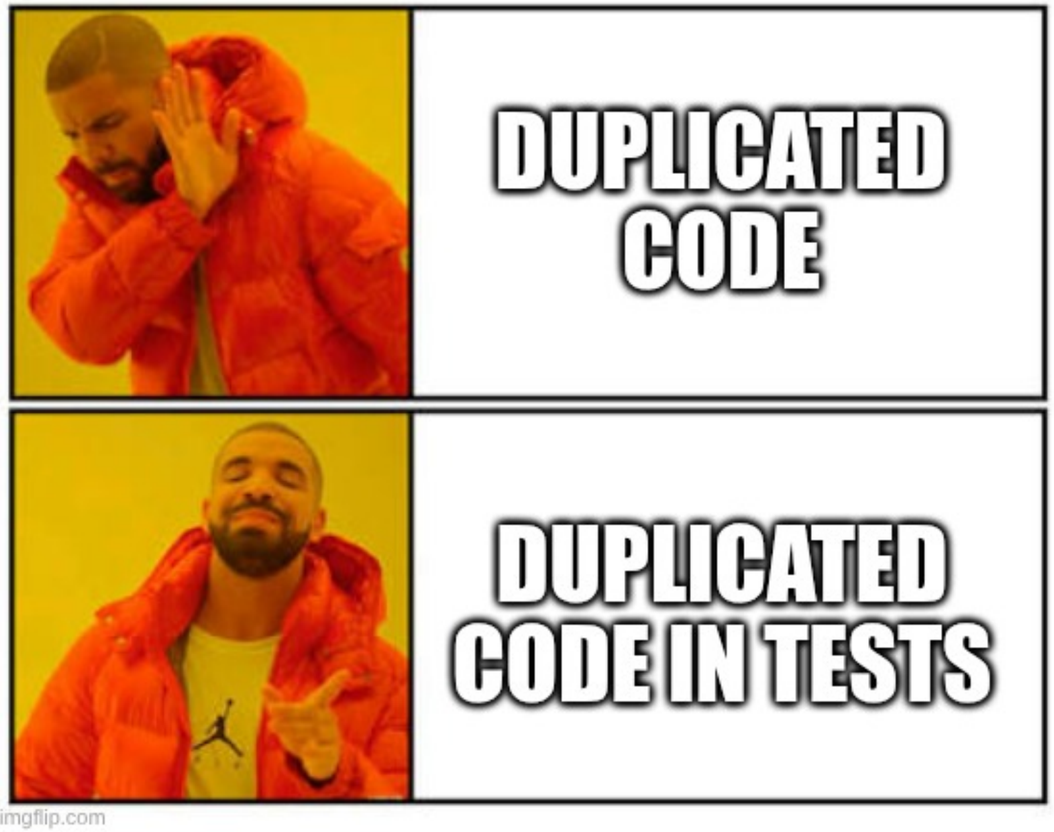
```
// Default repo, default PR
ctx.approve((), "karel");
```

```
// Default repo, default PR
ctx.approve((), "karel");

// Default repo, PR 2
ctx.approve(2, "karel");
```

```
// Default repo, default PR
ctx.approve((), "karel");

// Default repo, PR 2
ctx.approve(2, "karel");

// Repo foo/bar, PR 2
ctx.approve(("foo/bar", 2), "karel");
```

```rust
// Default repo, default PR
ctx.approve((), "karel");

// Default repo, PR 2
ctx.approve(2, "karel");

// Repo foo/bar, PR 2
ctx.approve(("foo/bar", 2), "karel");

// Custom reviewer ID
ctx.approve(
    ("foo/bar", 2),
    Reviewer::new("karel").id(1002)
);
```

Code reviewers:



Tip 2:

Test public (rather than private) interfaces

Tip 2:

Test public (rather than private) interfaces

a.k.a. Test behavior, not implementation

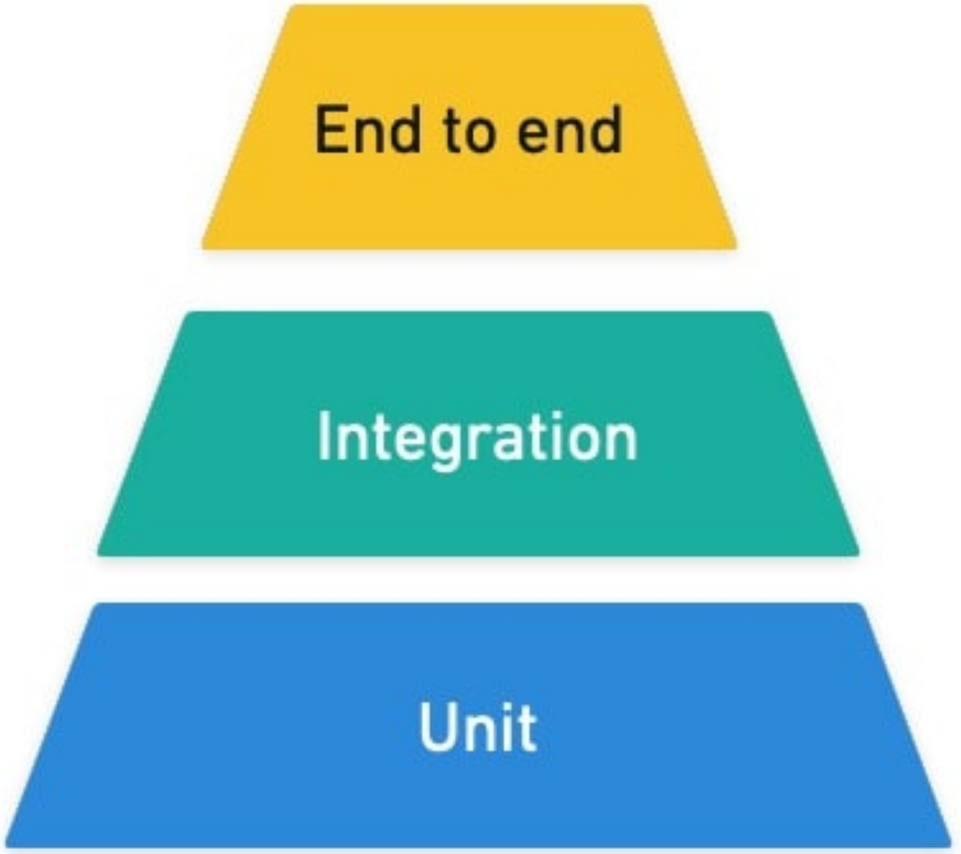
Tip 2:

Test public (rather than private) interfaces

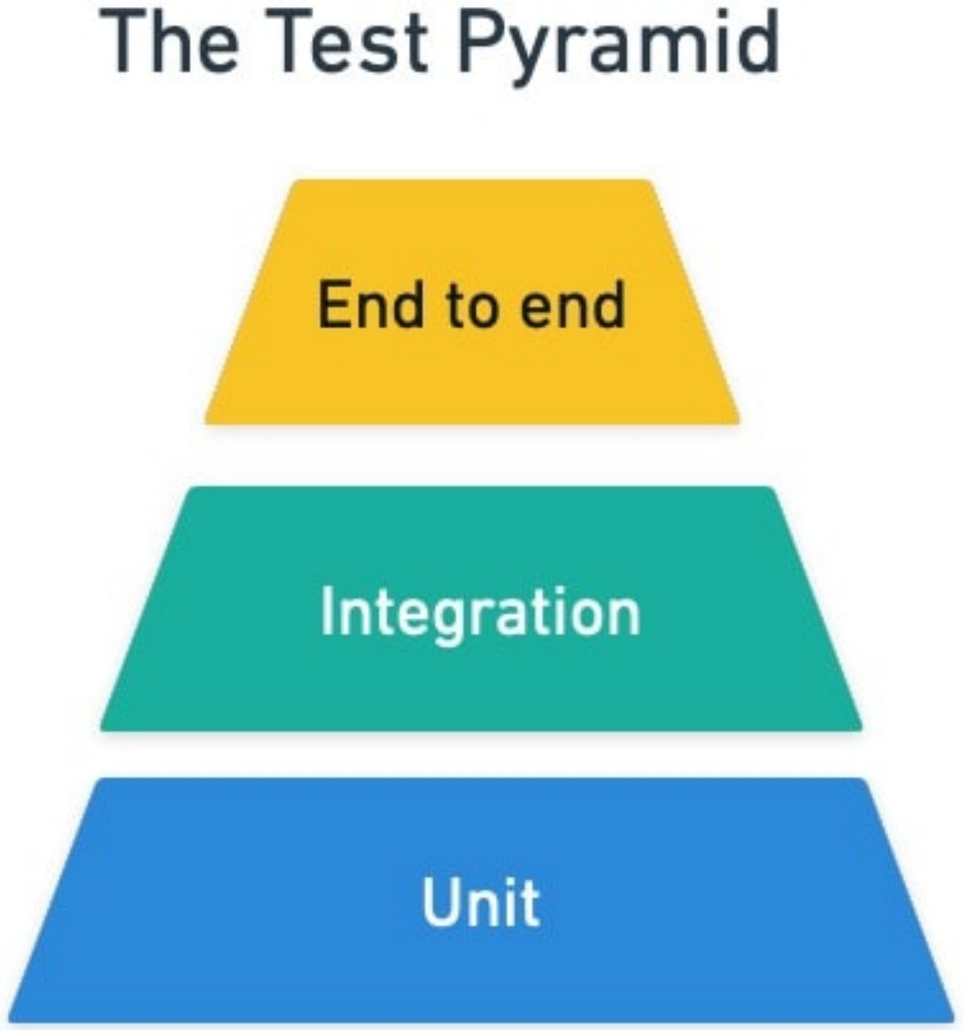
a.k.a. Test behavior, not implementation

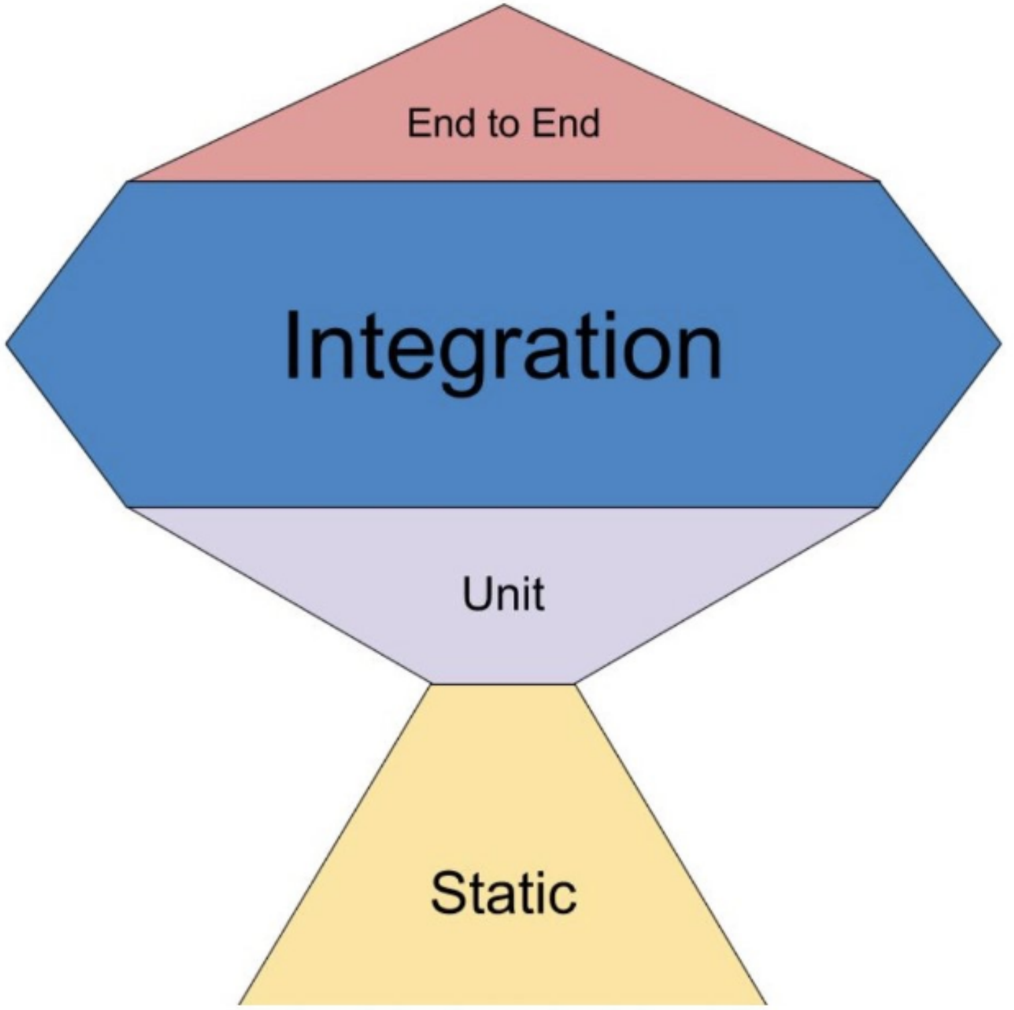
a.k.a. Prefer black-box (rather than white-box) tests

The Test Pyramid



Source: semaphore.io, Kent C. Dodds





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)

bors

```rust
#[test]
fn test_try_build() {
    let ctx = // create test context
    ctx.post_comment("@bors r=karel");
    let comment = ctx.get_comment();
    assert_eq!(comment, "Pull request was approved by karel");









}
```

bors

```rust
#[test]
fn test_try_build() {
    let ctx = // create test context
    ctx.post_comment("@bors r=karel");
    let comment = ctx.get_comment();
    assert_eq!(comment, "Pull request was approved by karel");

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



}
```

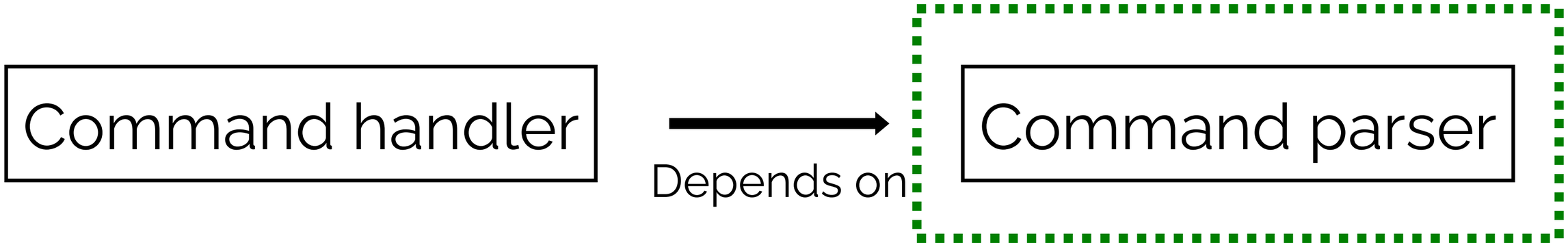
bors

```rust
#[test]
fn test_try_build() {
    let ctx = // create test context
    ctx.post_comment("@bors r=karel");
    let comment = ctx.get_comment();
    assert_eq!(comment, "Pull request was approved by karel");

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

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

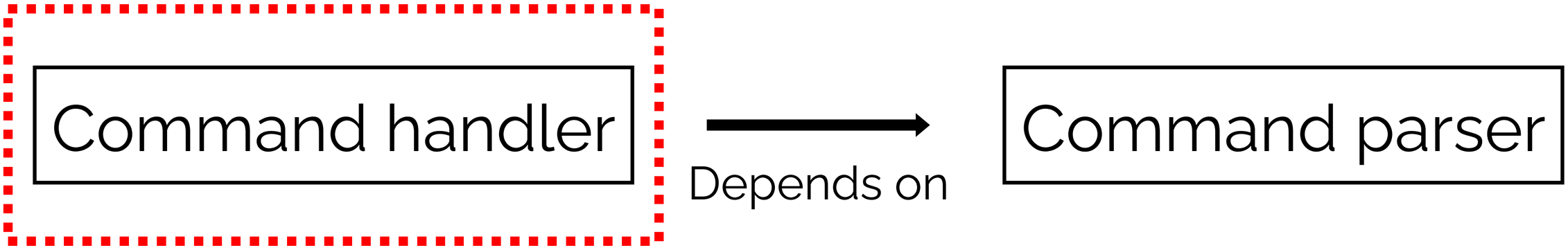
bors

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

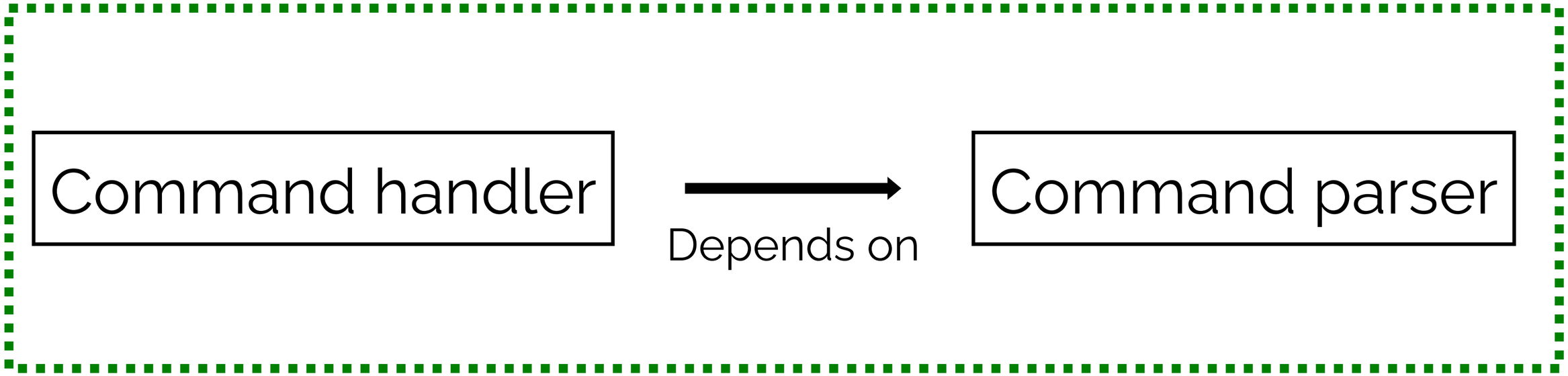

Command handler
Depends on
Command parser



Parser tests



Handler tests



Handler tests

Annoyance 2:

Tests are hard to understand

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

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

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

```rust
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

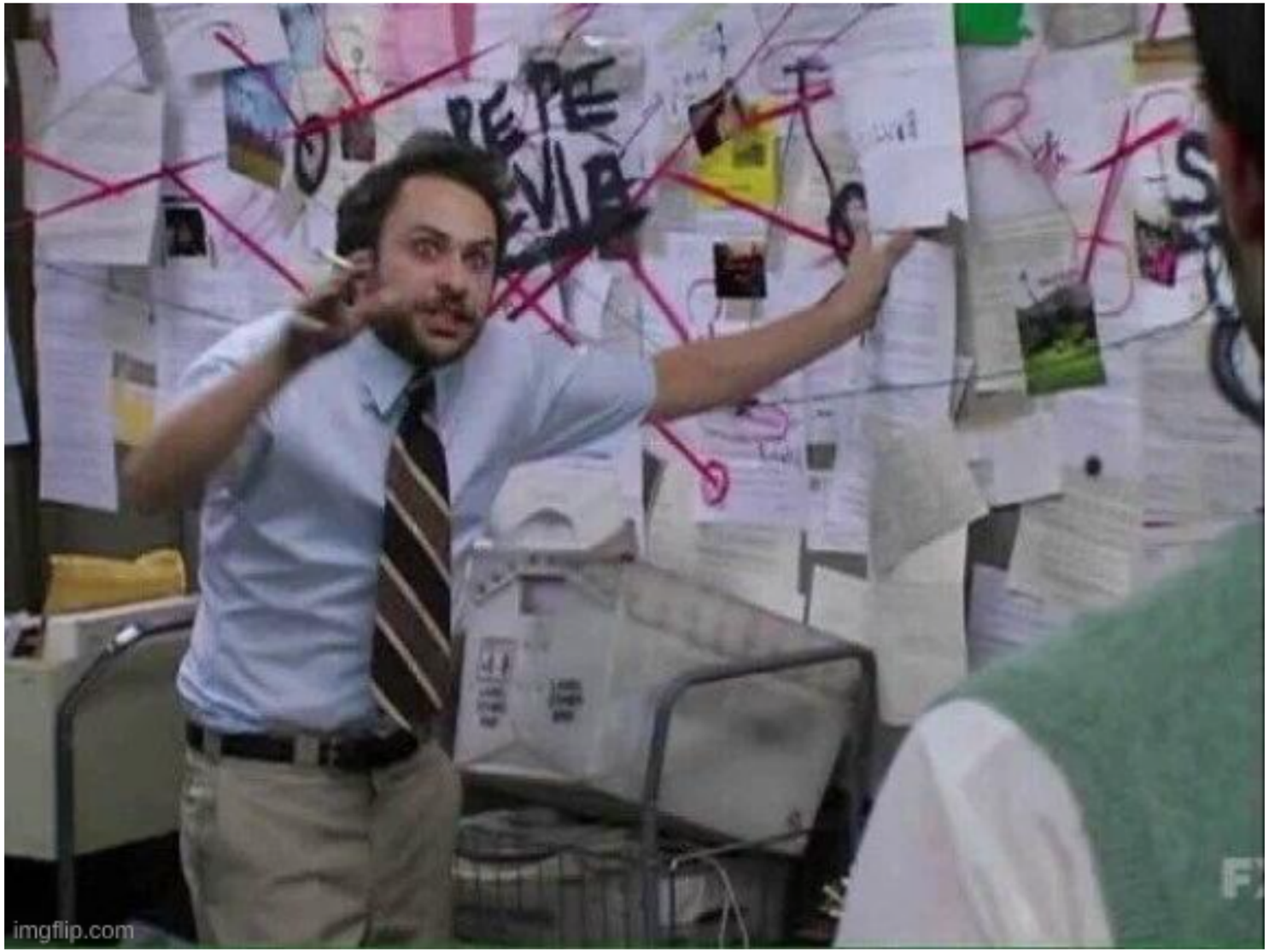
`VecDeque` currently allocates one extra empty element because it needs to discern between an empty and full queue. Besides wasting memory, this means `VecDeque::new` cannot currently be `const`.

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`.

```rust
#[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 are implementing an IDE

Imagine you are implementing an IDE

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

IntelliJ Rust/RustRover

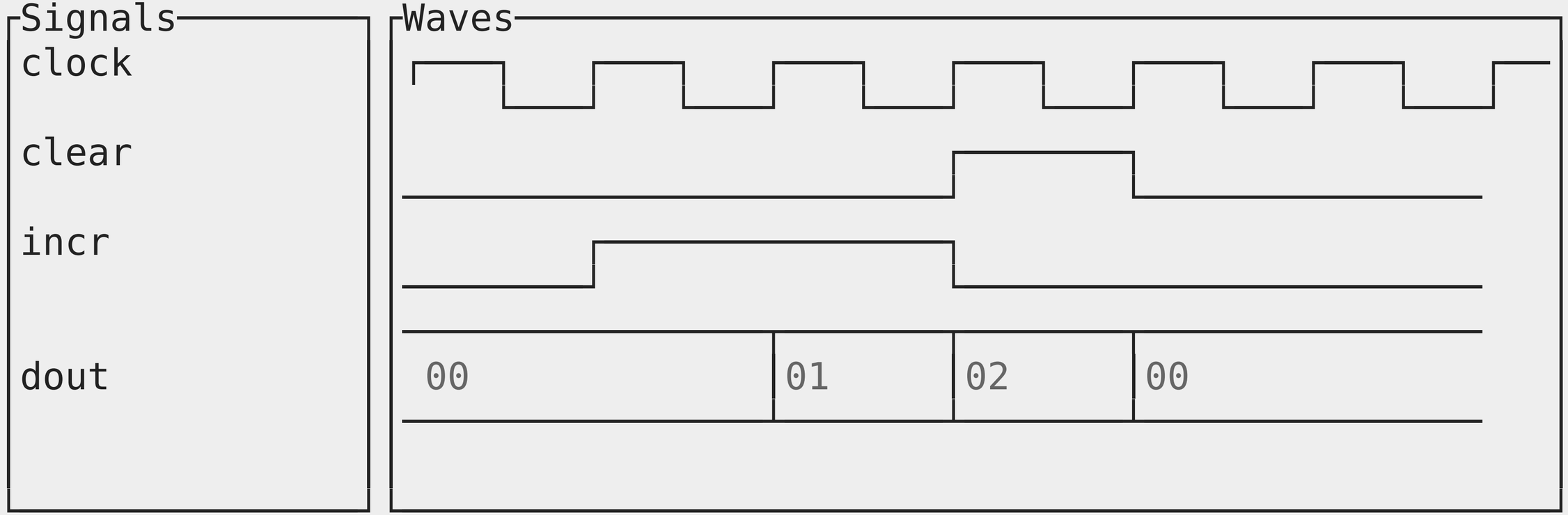
```
fun `test type alias`() = checkByCode("""
    type Foo = usize;
         //X

    trait T { type O; }

    struct S;

    impl T for S { type O = Foo; }
                            //^
""")
```

bootstrap (Rust compiler build system)

```
#[test]
fn cross_compile_test_library_stage_2() {
    let ctx = TestCtx::new();
    assert_eq!(
        ctx.config("test")
            .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>
    [test]  std 2 <target1>
"#);
}
```

Hardcaml test suite

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




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

Source: https://blog.janestreet.com/using-ascii-waveforms-to-test-hardware-designs

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 blessable

Tip 4:

Make tests data-driven

a.k.a. Make tests blessable

a.k.a. Use snapshot/expect tests

bootstrap (Rust compiler build system)

```
#[test]
fn cross_compile_test_library_stage_2() {
    let ctx = TestCtx::new();
    assert_eq!(
        ctx.config("test")
            .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>
    [test]  std 2 <target1>
"#);
}
```

bootstrap (Rust compiler build system)

```
#[test]
fn cross_compile_test_library_stage_2() {
    let ctx = TestCtx::new();
    insta::assert_snapshot!(
        ctx.config("test")
            .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>
    [test]  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
```

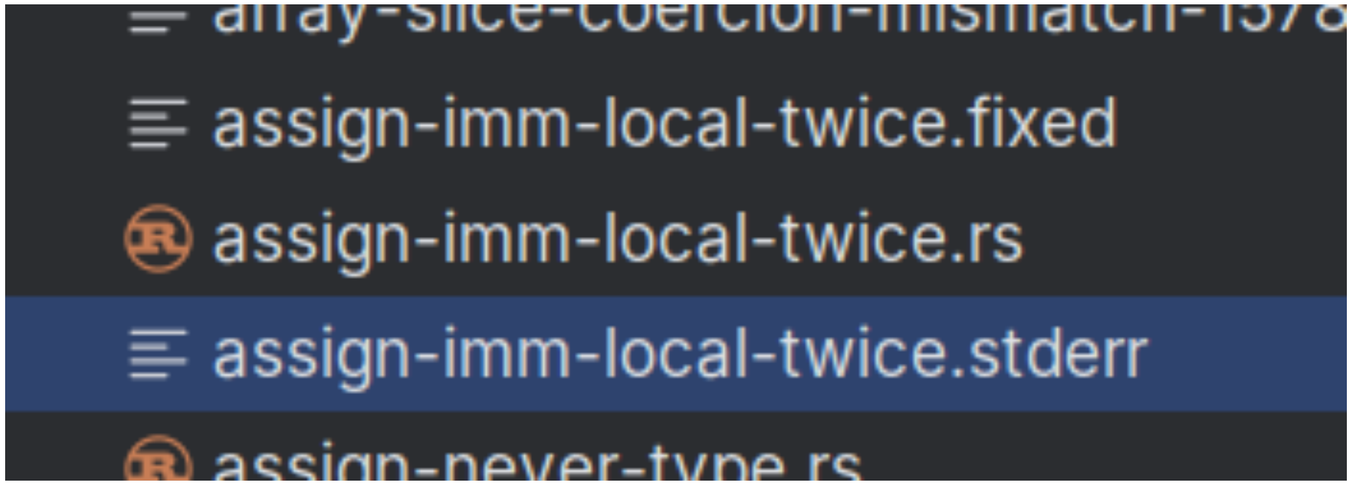
```
    0      0  [build] llvm <host>
    1      1  [build] rustc 0 <host> -> rustc 1 <host>
    2      2  [build] rustc 1 <host> -> std 1 <host>
           3 +[build] rustc 1 <host> -> rustc 2 <host>
    3      4  [check] rustc 2 <host> -> std 2 <host>
```

```
  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

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

insta::assert_snapshot!(
    ctx.get_comment(),
    @":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



Rust compiler UI test suite

assign-imm-local-twice.rs

```rust
//@ 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);
}
```

Rust compiler UI test suite

assign-imm-local-twice.stderr ✨

```
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`.
```

Rust compiler UI test suite

assign-imm-local-twice.fixed ✨

```rust
//@ 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
+7 -4
Viewed
@@ -2,10 +2,13 @@ error[E0121]: the placeholder `_` is not allowed within types on item signatures
--> $DIR/E0121.rs:1:13
LL | fn foo() -> _ { 5 }
^
|
not allowed in type signatures
help: replace with the correct return type: `i32`
^ not allowed in type signatures
help: replace with the correct return type
LL - fn foo() -> _ { 5 }
LL + fn foo() -> i32 { 5 }
error[E0121]: the placeholder `_` is not allowed within types on item signatures for static variables
--> $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


+2,578 −10

```python
@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)
```


Arthas
This entire city must be purged.
TEST SUITE
749 × 663

```
struct BorsService {
    db: Postgres
}
```

```
struct BorsService {
    db: Postgres
}

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

```
trait Database { … }
```

```
trait Database { … }

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

```rust
trait Database { … }

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

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

```rust
trait Database { … }

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

```rust
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 { … }
```

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

#[test]
fn test_bors() {

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

}
```

```rust
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`
```

bors

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

Isn't that slow?

Isn't that slow?

~400 integration tests in bors:

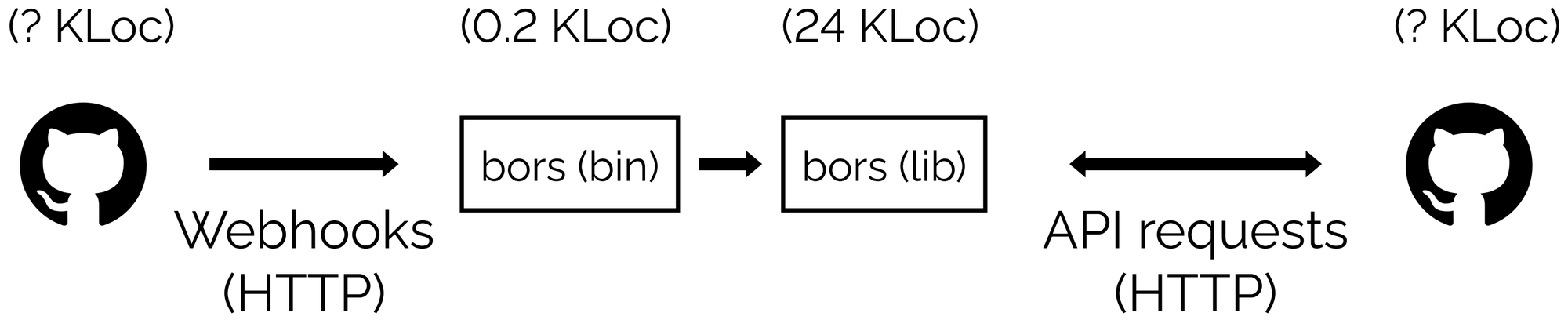
- ~20s (debug)
- ~8s (release)

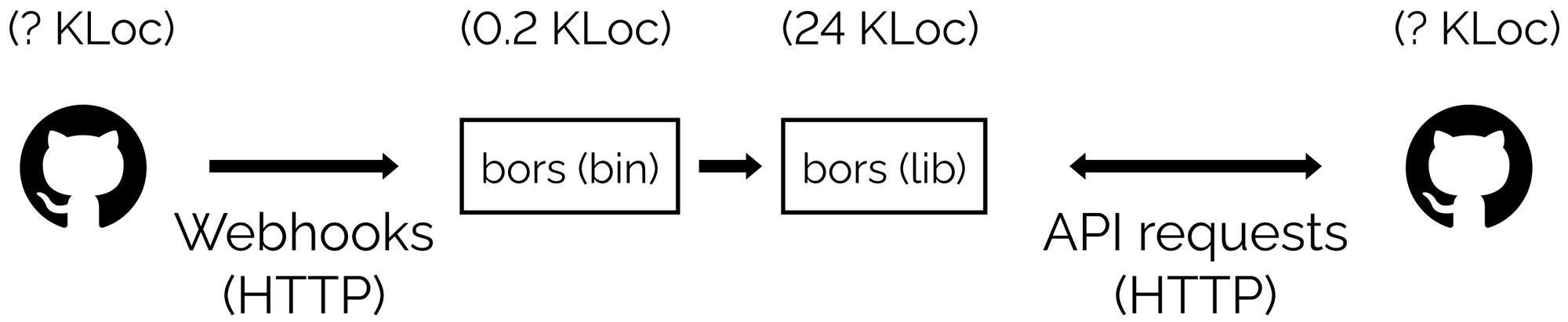
```
services:
  db:
    image: postgres:16.9
```

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

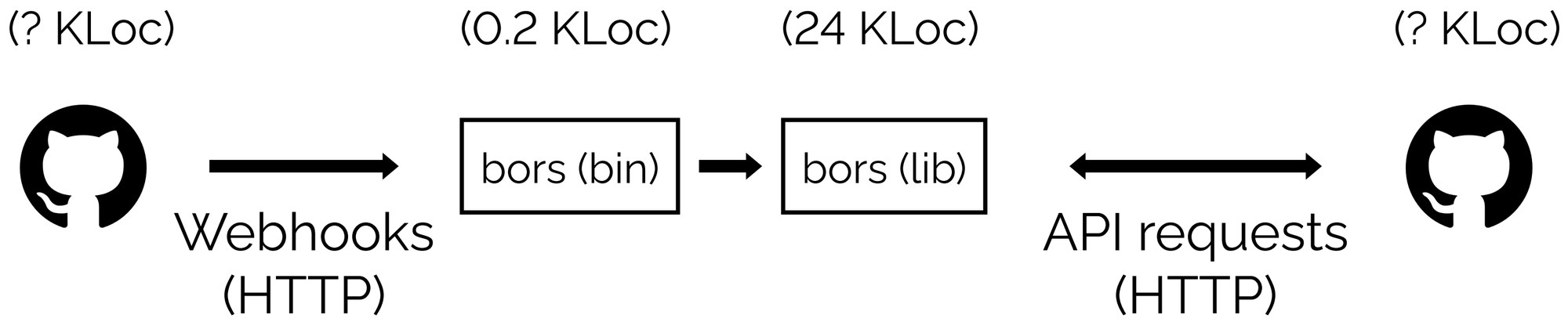
```
services:
  db:
    image: postgres:16.9
    # Turn off durability
    command: -c fsync=off
    # Use RAMdisk for storage
    tmpfs:
      - /var/lib/postgresql/data
```

Be pragmatic!

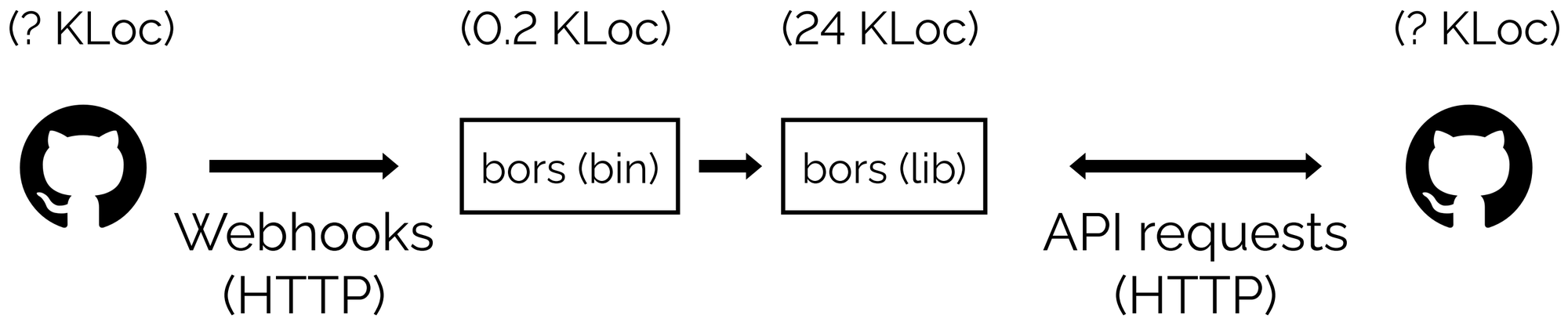

(? KLoc)
(0.2 KLoc)
(24 KLoc)
(? KLoc)
bors (bin)
bors (lib)
Webhooks
(HTTP)
API requests
(HTTP)



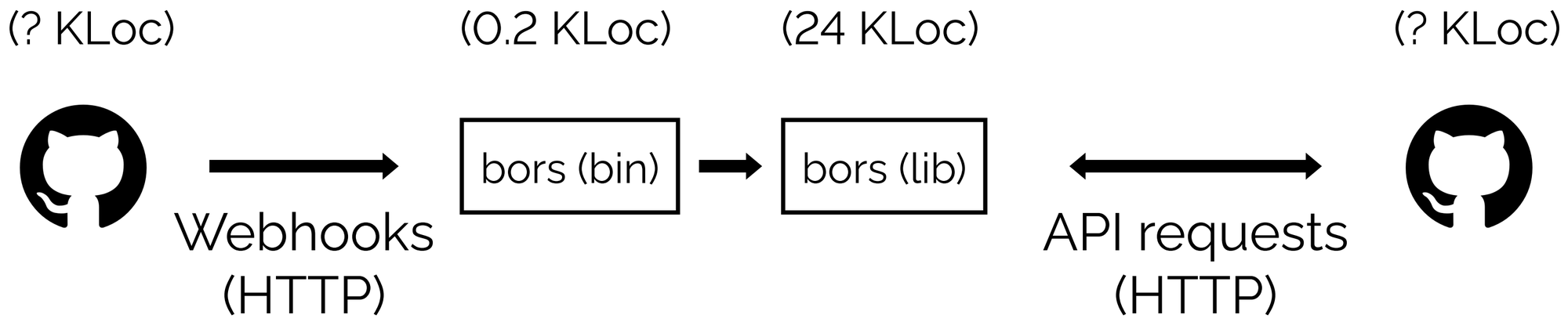
- Reimplement GitHub



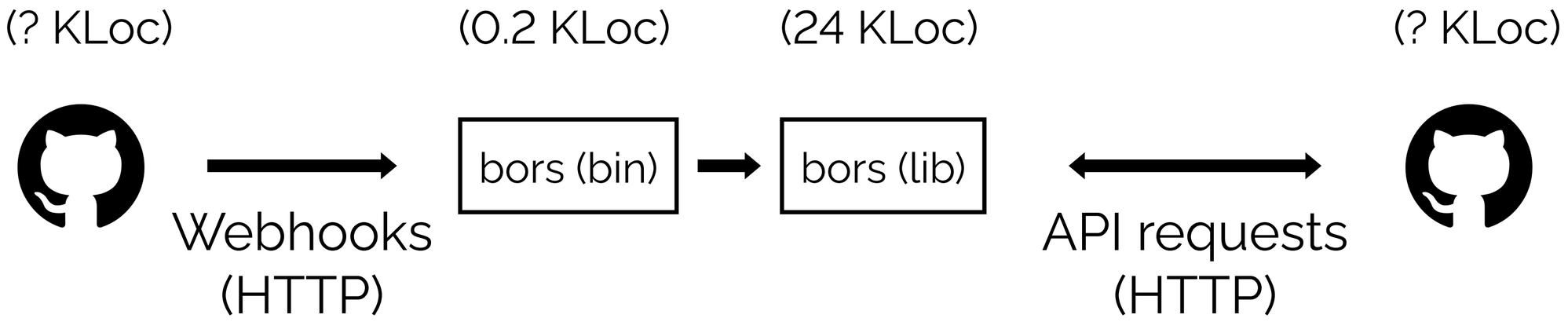
- Reimplement GitHub
- Deploy GitHub locally



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



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


(? KLoc)
(0.2 KLoc)
(24 KLoc)
(? KLoc)
bors (bin)
bors (lib)
Webhooks
(HTTP)
API requests
(HTTP)

Bors tests:


axum
bors (lib)
wiremock
Mocked webhooks
(HTTP)
API requests
(HTTP)

```
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:

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

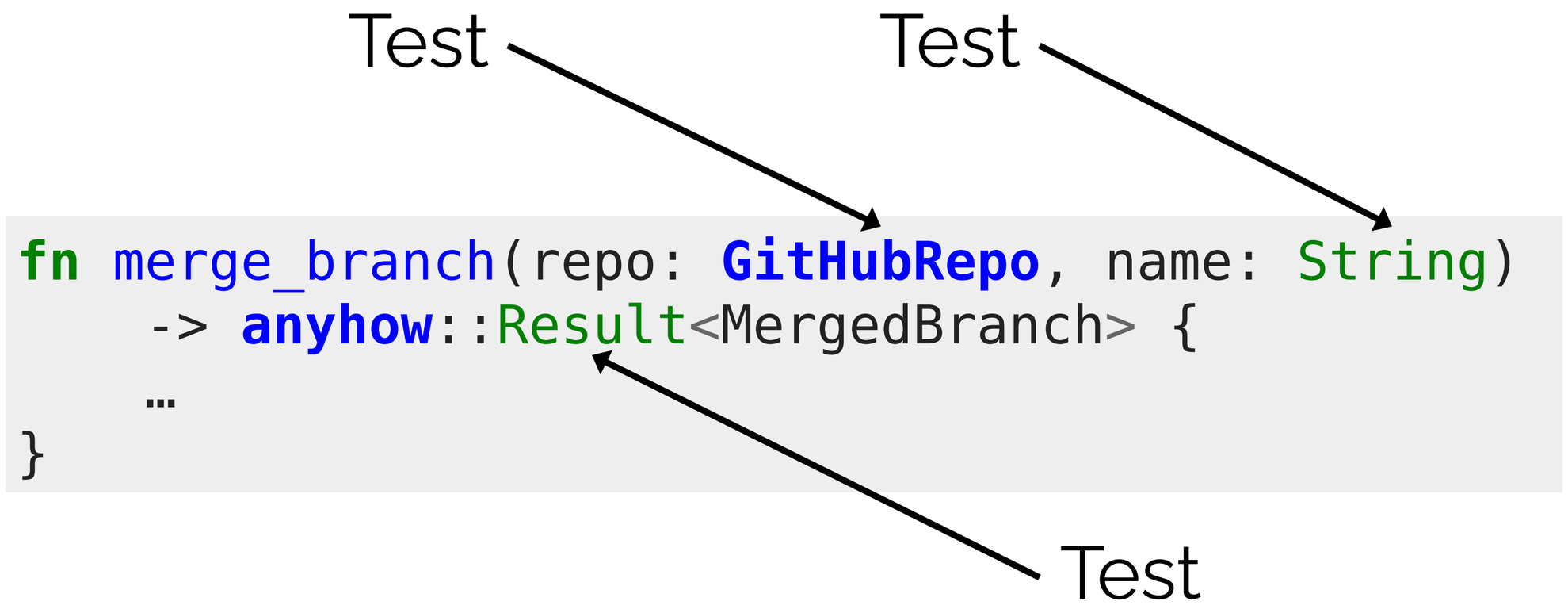
Feature flags

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

Tip 7:

Leverage compile-time tests

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


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

```
fn send_comment(
     repo: GithubRepo,
     number: PullRequestNumber,
     comment: Comment
) {
    …
}
```

```
fn attempt_merge(
    branch_name: &str,
    head_sha: &CommitSha,
    base_sha: &CommitSha,
    merge_message: &str,
    _merge_lock_is_held: &ExclusiveLockProof,
) -> anyhow::Result<MergeResult> {
    …
}
```

```rust
fn attempt_merge(
    branch_name: &str,
    head_sha: &CommitSha,
    base_sha: &CommitSha,
    merge_message: &str,
    _merge_lock_is_held: &ExclusiveLockProof,
) -> anyhow::Result<MergeResult> {
    …
}
```

bors

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

bors

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

```
error: error returned from database: column "state" of relation "build" does not exist
   --> src/database/operations.rs:529:24
    |
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 | |         )
    | |_________^
```

Rust compiler

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

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

Rust compiler

```rust
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:

Use asserts liberally

bors

```rust
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 9:

You can test (almost) anything


Kobzol approved these changes on Mar 14
View reviewed changes
Kobzol left a comment
Member
LGTM!

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

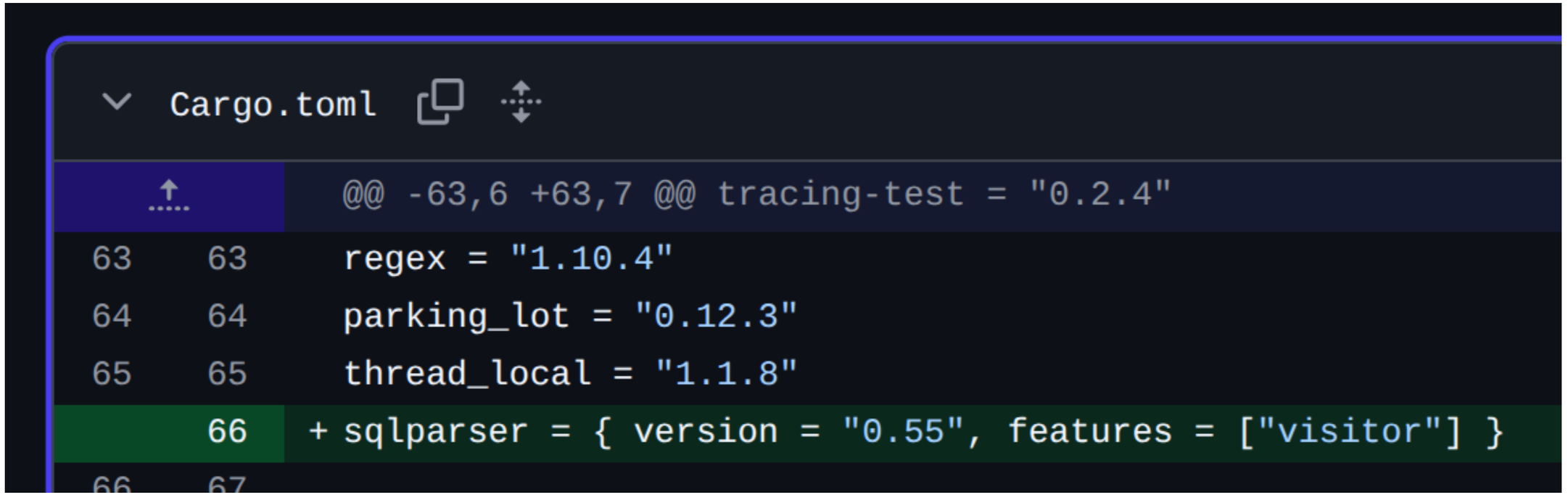
> Caution
>
> When adding a new `NOT NULL` column, always specify the `DEFAULT` value that will be backfilled during the migration! Otherwise, the migration might break the deployed bors service.

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

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









}
```

```rust
#[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();









        }
    }
}
```

```rust
#[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: {error}",
                    file.path().display()
                );
            }
        }
    }
}
```

```rust
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(())
    }
}
```

```rust
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);
```

```rust
#[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(())
}
```

```rust
#[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(())
}
```

```rust
#[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
        );
    }
}
```

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?

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

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

async fn background_job() {

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

```rust
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;
    }
}
```

```rust
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;
    }
}
```

```rust
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
    }
}
```

```rust
async fn background_job(mut rx: Receiver<()>) {

    while let Some(msg) = rx.recv().await {
        do_work().await;
    }
}
```

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

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

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

```rust
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;
}
```

```rust
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, …

bors

```
async fn unapprove() {
    …
    ctx.post_comment("@bors r-").await?;




}
```

bors

```rust
async fn unapprove() {
    …
    ctx.post_comment("@bors r-").await?;

    // Bot comment acts as a synchronization point
    insta::assert_snapshot!(
        ctx.get_next_comment().await?,
        @"Commit pr-1-sha has been unapproved."
    );


}
```

bors

```rust
async fn unapprove() {
    …
    ctx.post_comment("@bors r-").await?;

    // Bot comment acts as a synchronization point
    insta::assert_snapshot!(
        ctx.get_next_comment().await?,
        @"Commit pr-1-sha has been unapproved."
    );

    ctx.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

bors

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

bors

```rust
#[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)]
        WAIT_FOR_WORKFLOW_STARTED.mark();
    }
    …
}
```

bors

```rust
#[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)]
        WAIT_FOR_WORKFLOW_STARTED.mark();
    }
    …
}

async fn test(ctx: TestCtx) {
    ctx.start_workflow().await;
    WAIT_FOR_WORKFLOW_STARTED.sync().await;
    ctx.assert_workflow_started().await;
}
```

Tip 11:

Collect all errors during tests

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

    Ok(())
}
```

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

    Ok(())
}
```

Error: Connection reset by peer (os error 104)

```rust
async fn test_approve() -> anyhow::Result<()> {
    let ctx = start_bors().await;
    ctx.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 …

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









}
```

```rust
async fn run_test<F>(test_fn: F) -> anyhow::Result<()>
where
    F: AsyncFnOnce(&mut TestCtx) -> 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)
}
```

```rust
async fn run_test<F>(test_fn: F) -> anyhow::Result<()>
where
    F: AsyncFnOnce(&mut TestCtx) -> 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 |ctx| {
        ctx.post_comment("@bors r+").await?;
        Ok(())
    }).await;
}
```

Deterministic Simulation Testing

- TigerBeetle
- antithesis

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

        ctx
            .get_pr(())
            .await
            .expect_approved_by(&User::default_pr_author().name);
        Ok(())
    })
    .await;
}
```

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!

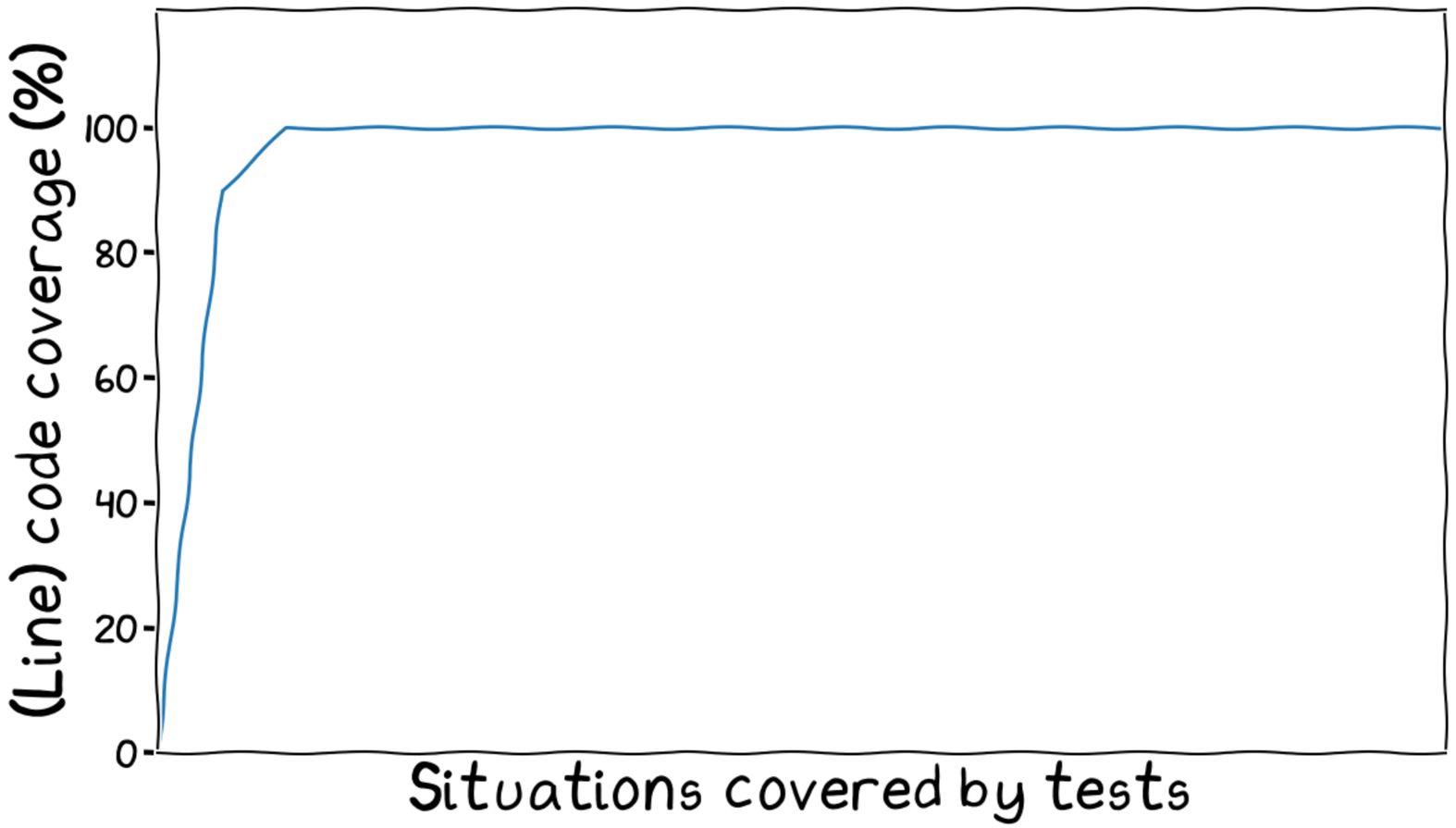
```
fn get_host_without_tld(mut text: &str) -> &str {
    if let Some(pos) = text.find("@") {
        text = &text[pos..];
    }
    &text[text.find(".").unwrap()..]
}
```

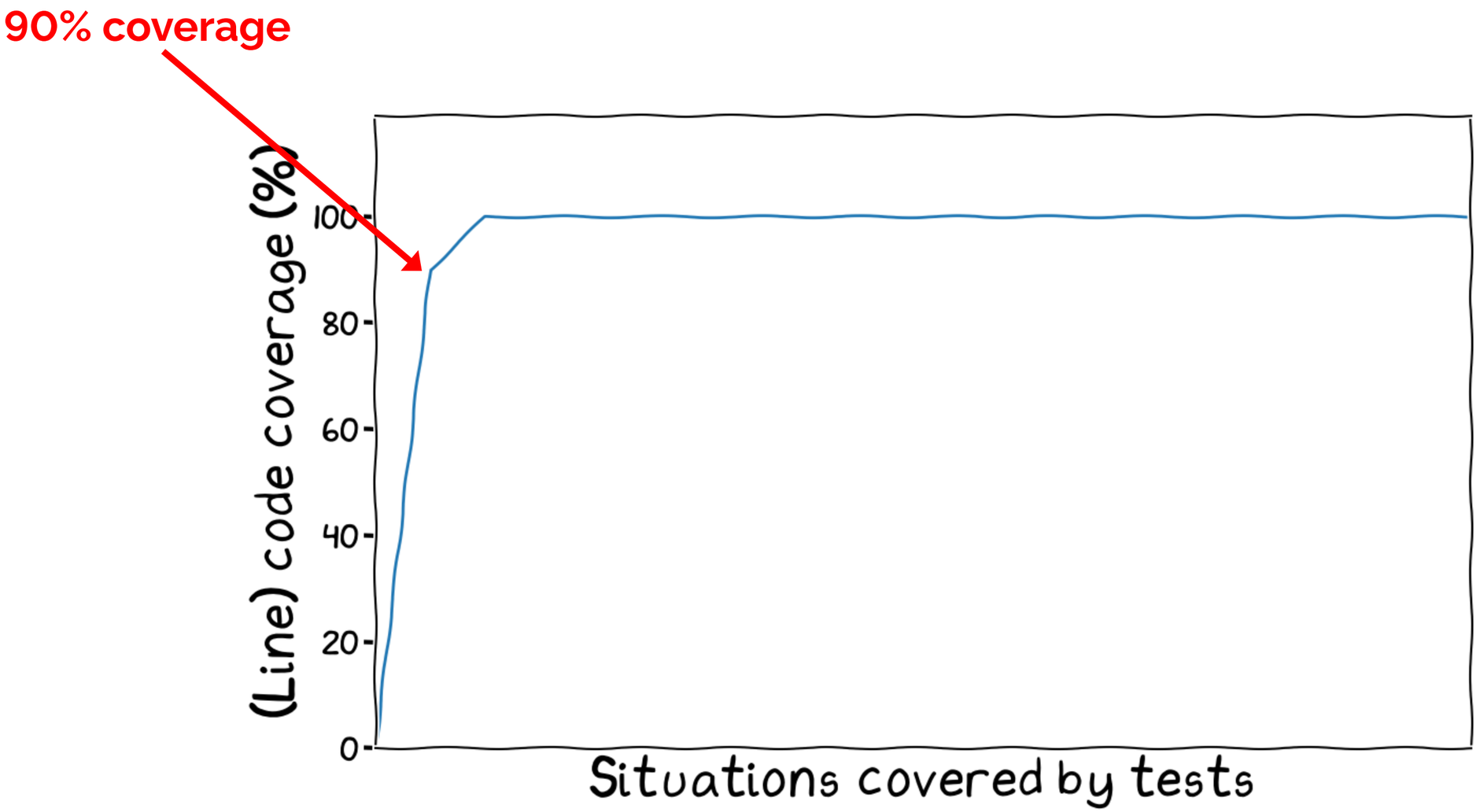
```rust
fn get_host_without_tld(mut text: &str) -> &str {
    if let Some(pos) = text.find("@") {
        text = &text[pos..];
    }
    &text[text.find(".").unwrap()..]
}

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



}
```

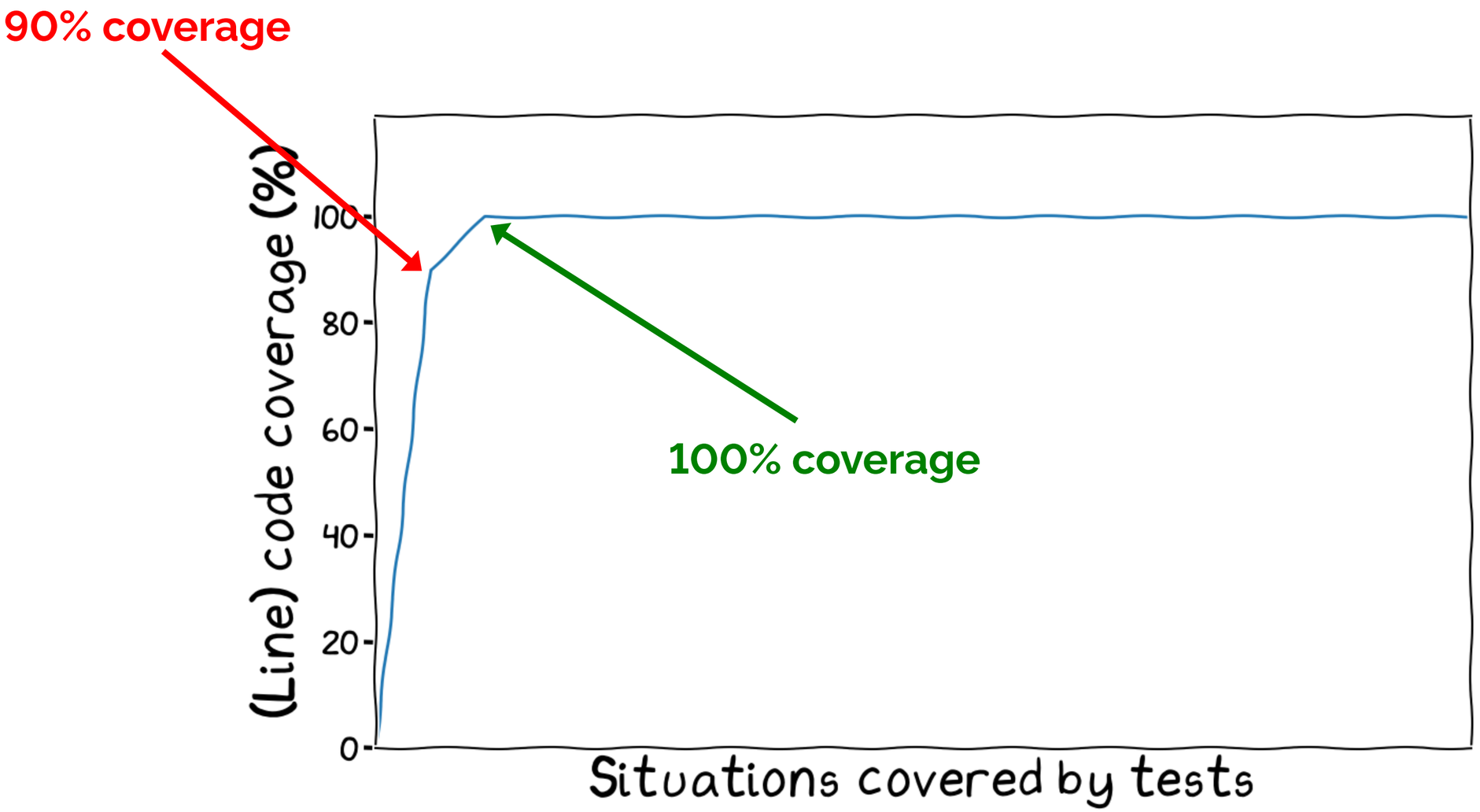
```rust
fn get_host_without_tld(mut text: &str) -> &str {
    if let Some(pos) = text.find("@") {
        text = &text[pos..];
    }
    &text[text.find(".").unwrap()..]
}

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

}
```

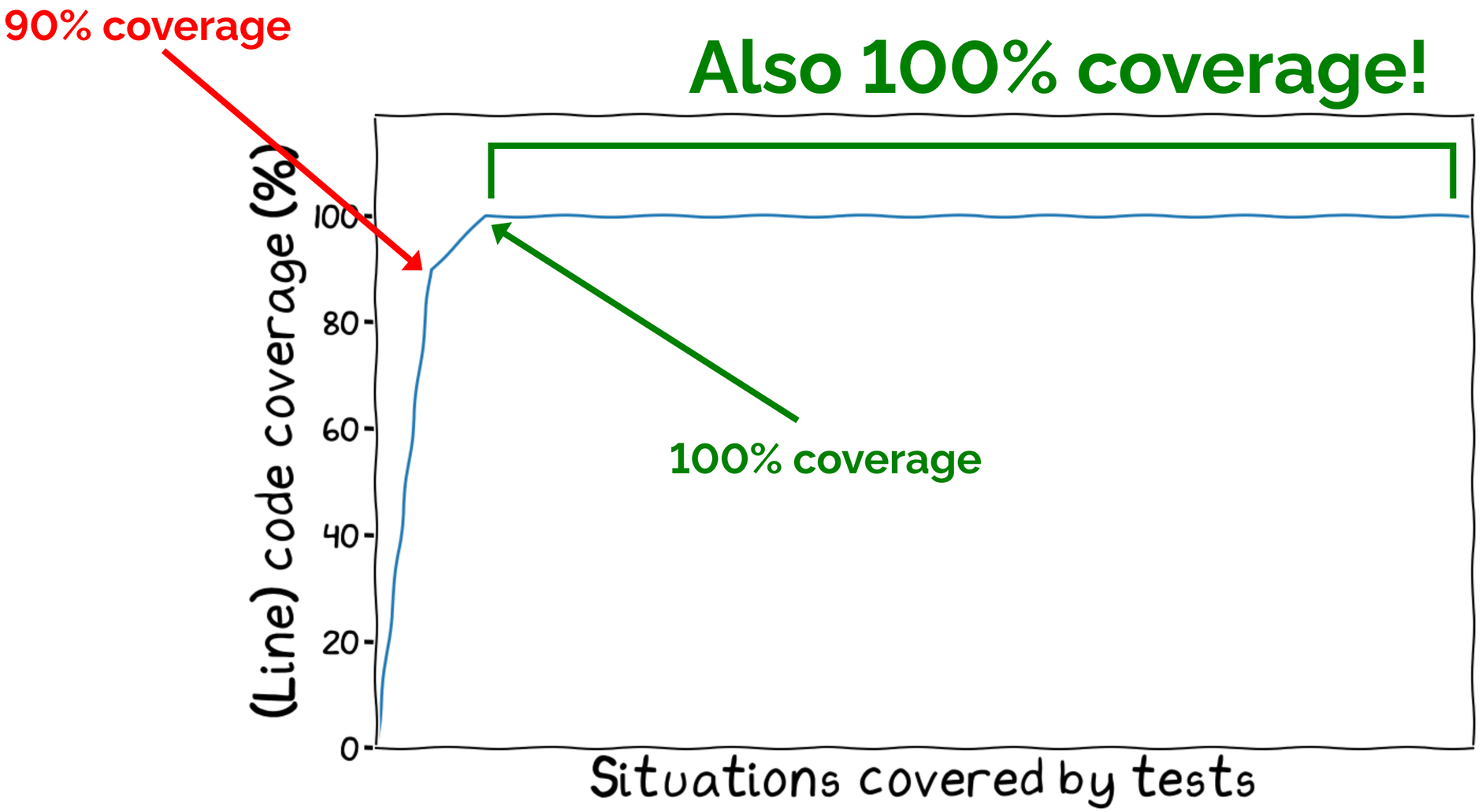
```rust
fn get_host_without_tld(mut text: &str) -> &str {
    if let Some(pos) = text.find("@") {
        text = &text[pos..];
    }
    &text[text.find(".").unwrap()..]
}

#[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 :-(
}
```


(Line) code coverage (%)
100
80
60
40
20
0
Situations covered by tests


(Line) code coverage (%)
100
80
60
40
20
0
Situations covered by tests


90% coverage
(Line) code coverage (%)
100
80
60
40
20
0
Situations covered by tests


90% coverage
(Line) code coverage (%)
100
80
60
40
20
0
100% coverage
Situations covered by tests


90% coverage
Also 100% coverage!
100% coverage
(Line) code coverage (%)
0
20
40
60
80
100
Situations covered by tests

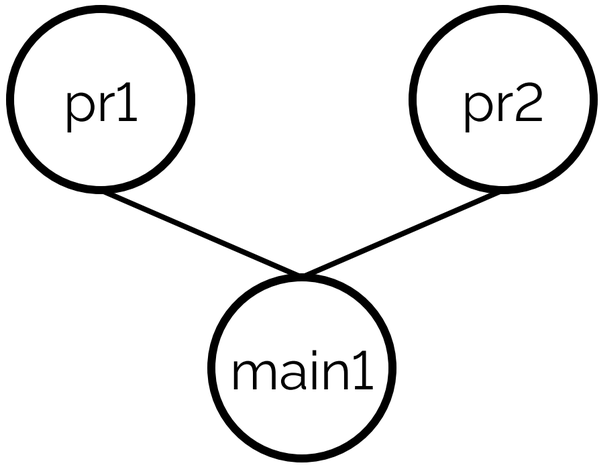
Tip 12:

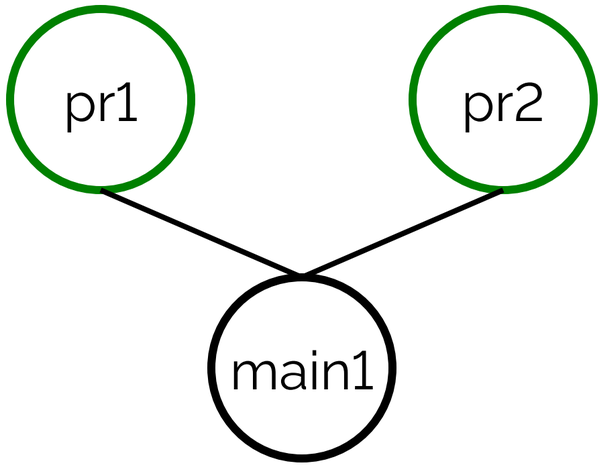
Metrics can be useful
But don't worry about them too much

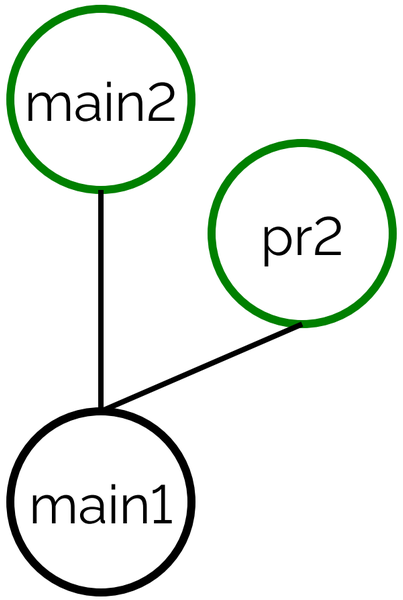
Annoyance 7:

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) ”


pr1
pr2
main1


pr1
pr2
main1


main2
pr2
main1


main3
main2
main1

Tip 13:

Use merge queues/trains

Tip 14:

Invest in test infrastructure

Tip 14:

Invest in test infrastructure

It is 💯 worth it

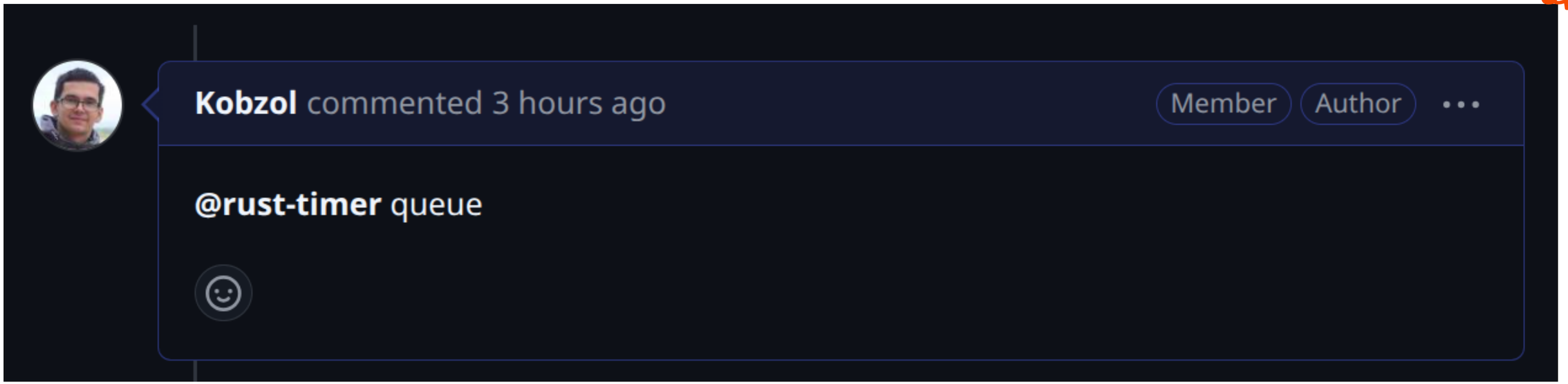
Tip 15:

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

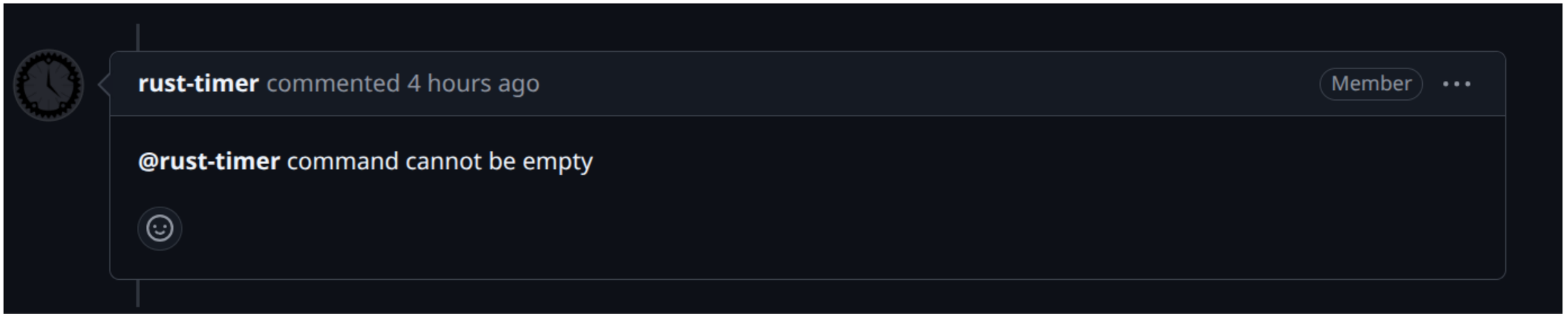
Kobzol commented 3 hours ago Member Author •••

@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: {error}\n")),
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
Member
@rust-timer command cannot be empty

rust-timer commented 4 hours ago Member

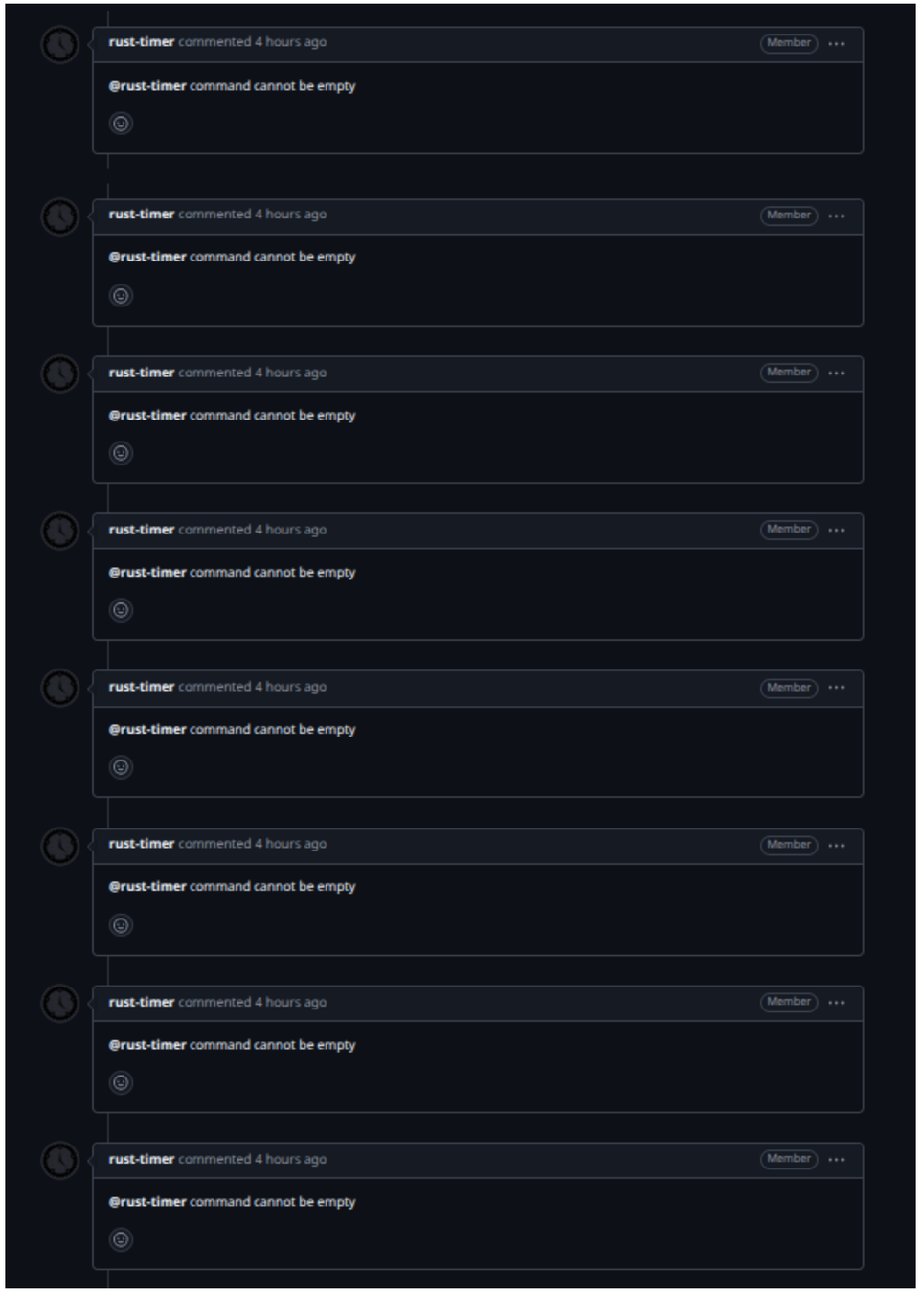
@rust-timer command cannot be empty

rust-timer commented 4 hours ago Member

@rust-timer command cannot be empty

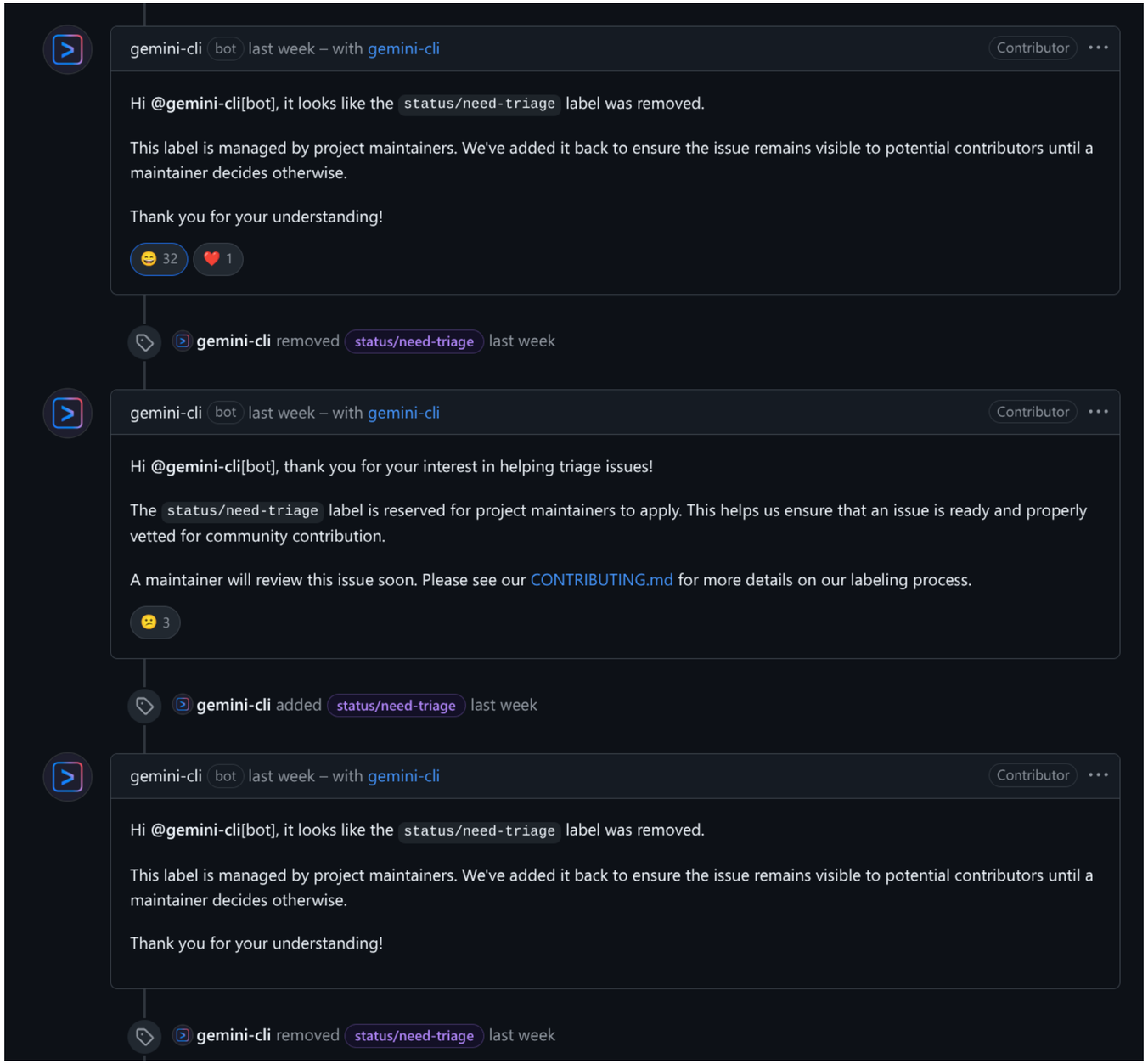
rust-timer commented 4 hours ago Member

@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
rust-timer commented 4 hours ago
Member
@rust-timer command cannot be empty

565 hidden items

Load more...



Source: https://github.com/google-gemini/gemini-cli/issues/16750

TLDR:

- Build high-level test APIs

TLDR:

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

TLDR:

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

Thank you for your attention!

Slides are available here:

Slides were programmed using `github.com/spirali/elsie`
in 2342 lines of Python

Several emojis were used from the Noto Emoji pack