MUTATION TESTING AT GOOGLE

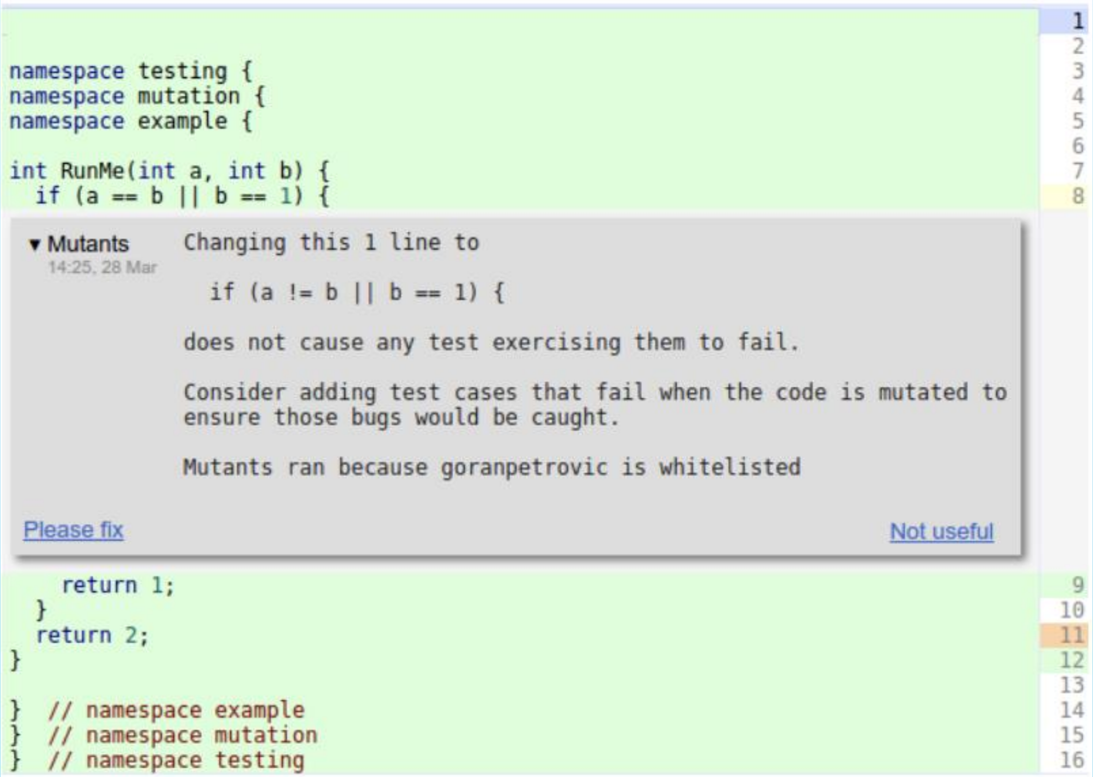
Omar Aguirre Rodríguez
Samuel de la Calle Fernández
Raúl Antuña Suárez
Pablo Rodríguez García
Carlos Sampedro Menéndez

WHAT IS MUTATION TESTING?

Mutation testing assesses test suite efficacy by inserting small faults into programs and measuring the ability of the test suite to detect them.

These faults are called mutants and simulate the bugs you could naturally introduce. Tests should detect these mutants correctly.

EXAMPLE



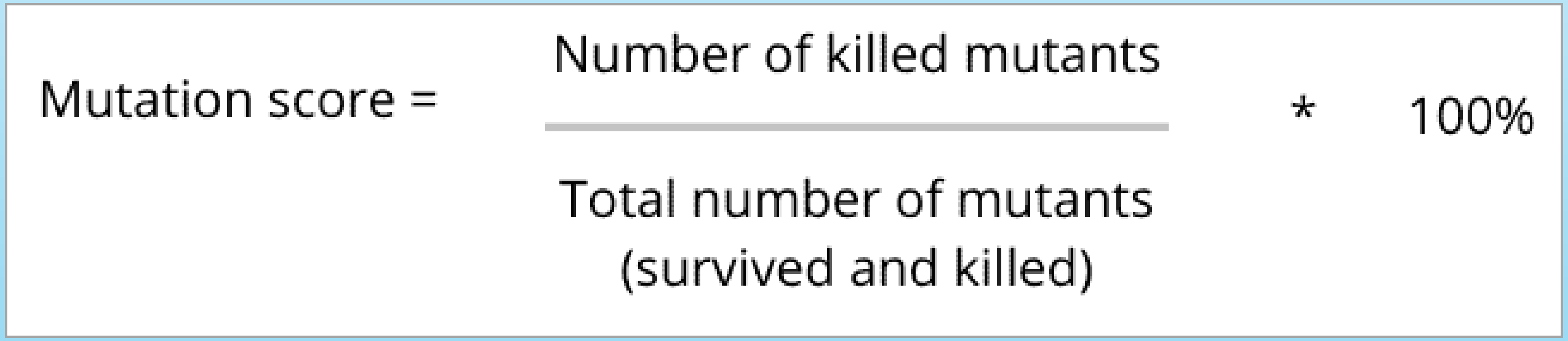
```


namespace testing {
namespace mutation {
namespace example {

int RunMe(int a, int b) {
  if (a == b || b == 1) {
```

```
Changing this 1 line to

  if (a != b || b == 1) {

does not cause any test exercising them to fail.

Consider adding test cases that fail when the code is mutated to
ensure those bugs would be caught.

Mutants ran because goranpetrovic is whitelisted
```

```
    return 1;
  }
  return 2;
}

}  // namespace example
}  // namespace mutation
}  // namespace testing
```

MUTATION TESTING SCORE

Not as well defined as line coverage.

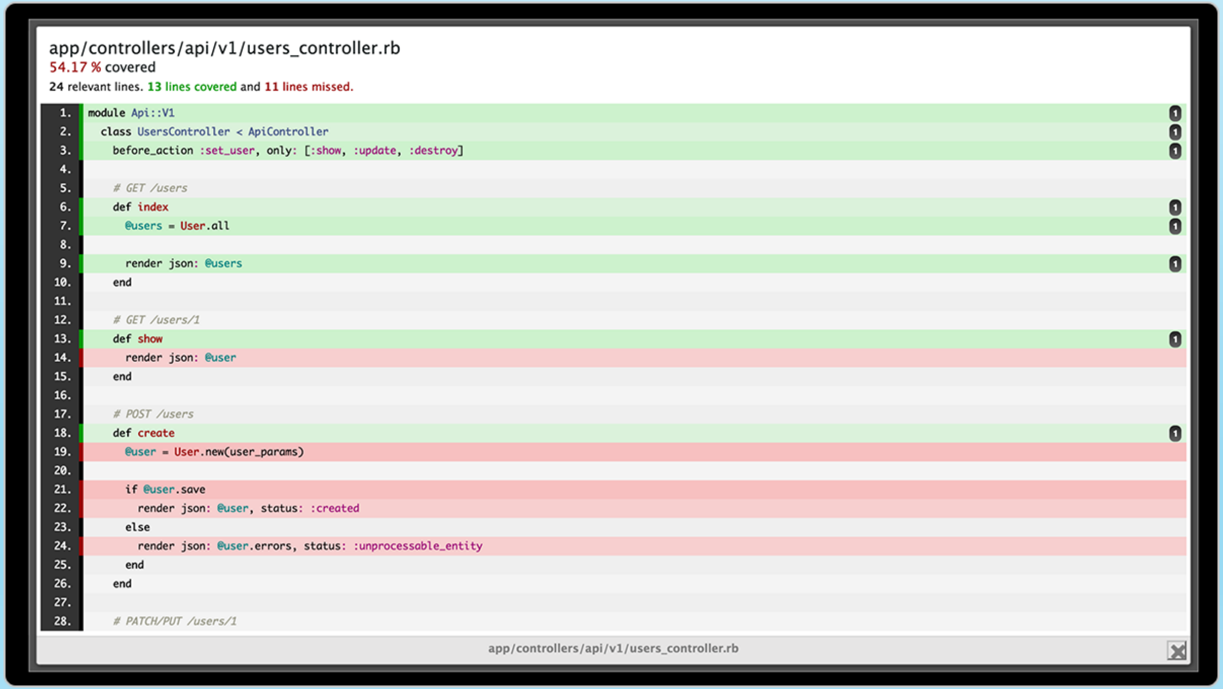


$$\text{Mutation score} = \frac{\text{Number of killed mutants}}{\text{Total number of mutants (survived and killed)}} * 100\%$$

DIFFERENCES WITH OTHER TESTING METRICS

Line coverage tells you if something is covered or not (objective)

Mutation testing score depends on the quality of the mutants (subjective)



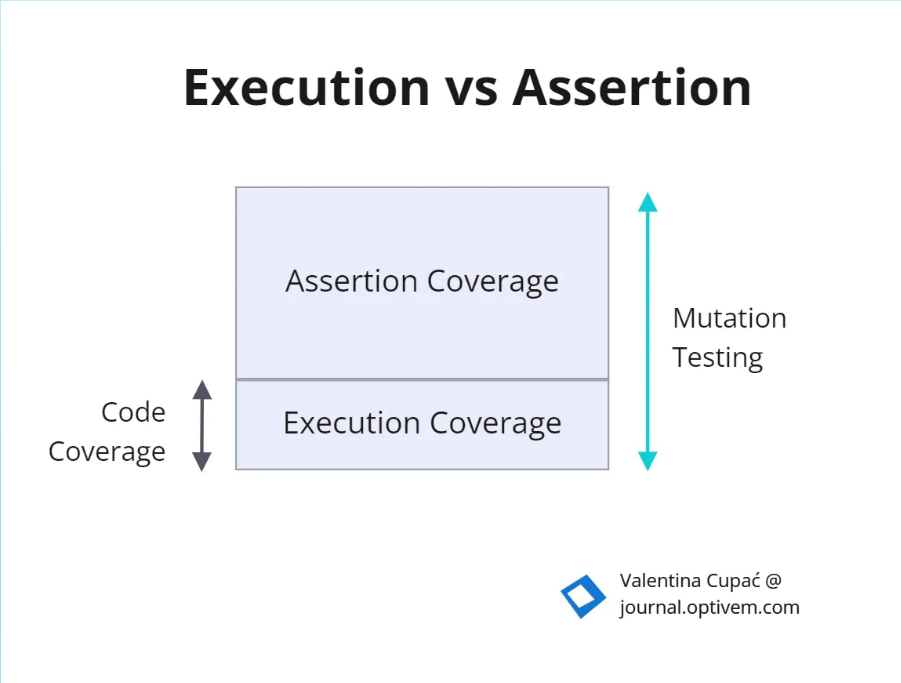
app/controllers/api/v1/users_controller.rb

54.17 % covered

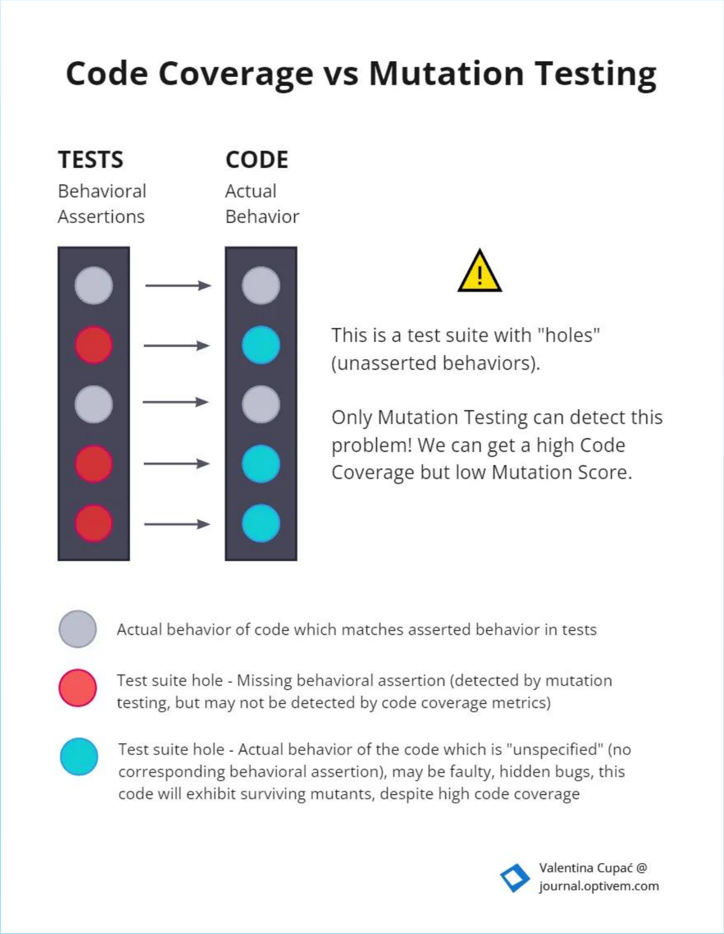
24 relevant lines. 13 lines covered and 11 lines missed.

```ruby
module Api::V1
  class UsersController < ApiController
    before_action :set_user, only: [:show, :update, :destroy]

    # GET /users
    def index
      @users = User.all

      render json: @users
    end

    # GET /users/1
    def show
      render json: @user
    end

    # POST /users
    def create
      @user = User.new(user_params)

      if @user.save
        render json: @user, status: :created
      else
        render json: @user.errors, status: :unprocessable_entity
      end
    end

    # PATCH/PUT /users/1
```

DIFFERENCES WITH OTHER TESTING METRICS



Execution vs Assertion



Code Coverage vs Mutation Testing

This is a test suite with "holes" (unasserted behaviors).

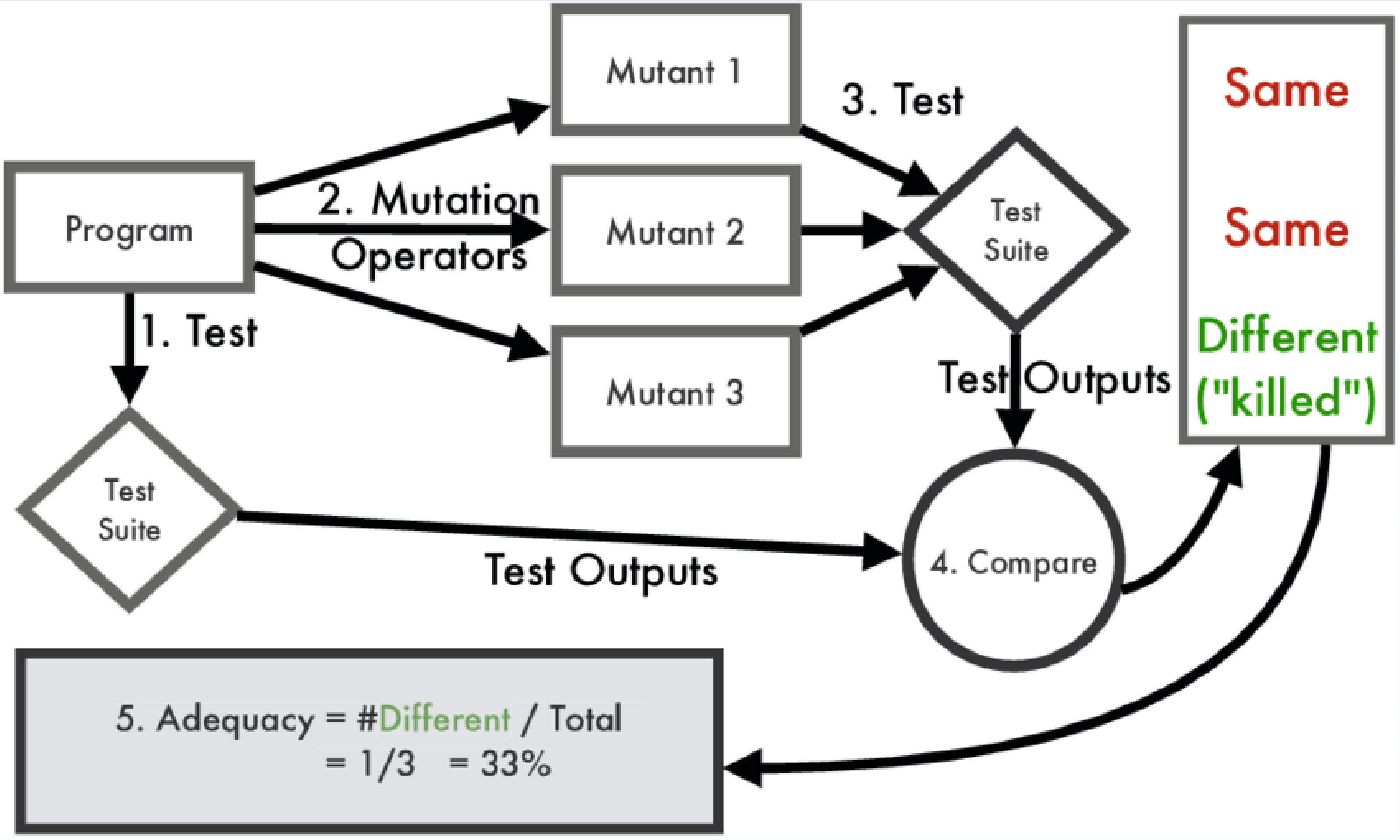
Only Mutation Testing can detect this problem! We can get a high Code Coverage but low Mutation Score.

● Actual behavior of code which matches asserted behavior in tests

● Test suite hole - Missing behavioral assertion (detected by mutation testing, but may not be detected by code coverage metrics)

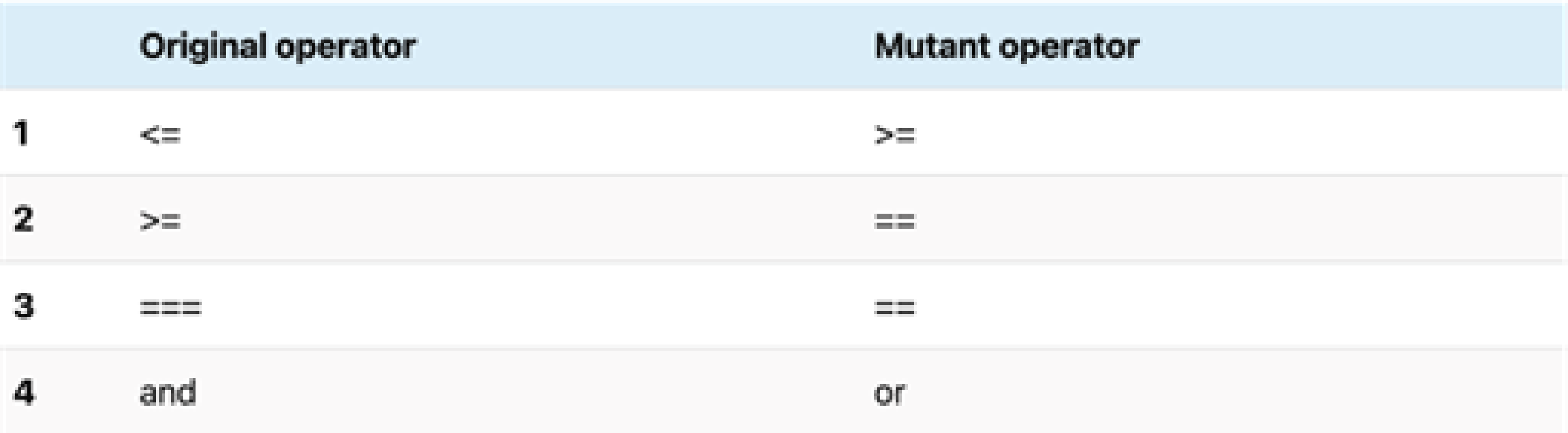
● Test suite hole - Actual behavior of the code which is "unspecified" (no corresponding behavioral assertion), may be faulty, hidden bugs, this code will exhibit surviving mutants, despite high code coverage

HOW CAN WE (ACTUALLY) TEST IT?



HOW CAN WE (ACTUALLY) TEST IT?

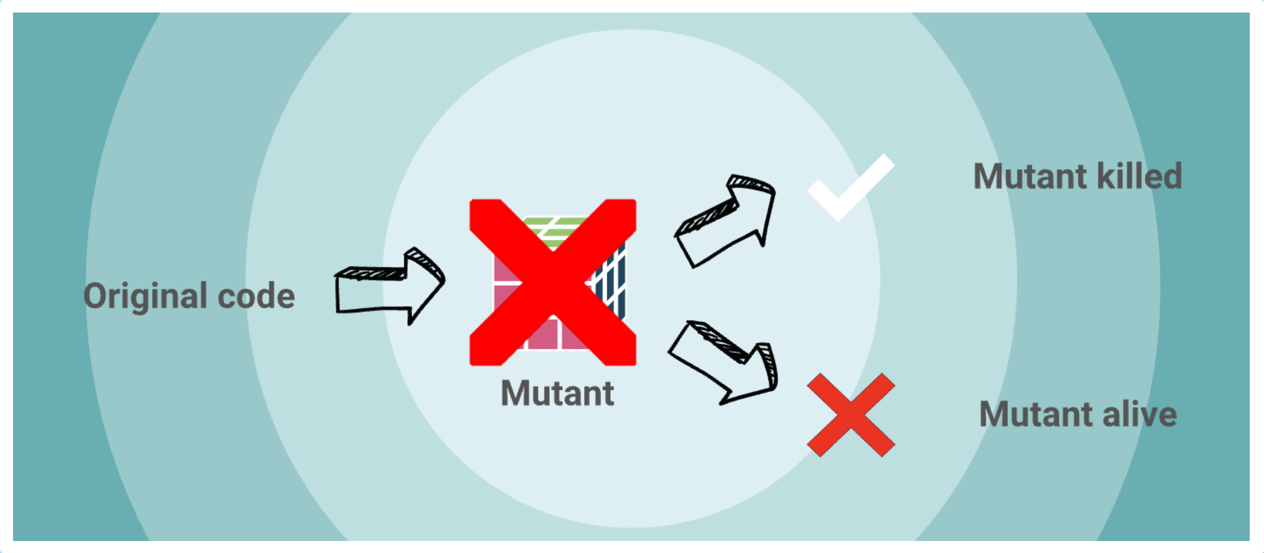
- It's posible to test functions/methods.



	Original operator	Mutant operator
1	<=	>=
2	>=	==
3	===	==
4	and	or

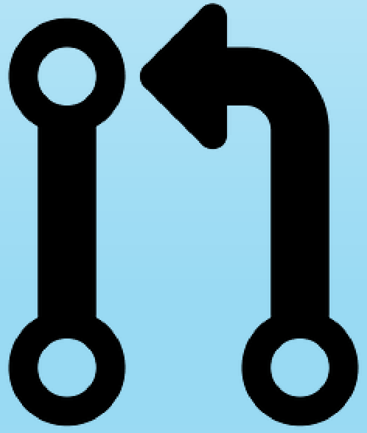
CHANGE DETECTOR TESTS

- Test **specific implementation details.** (usually minor ones)

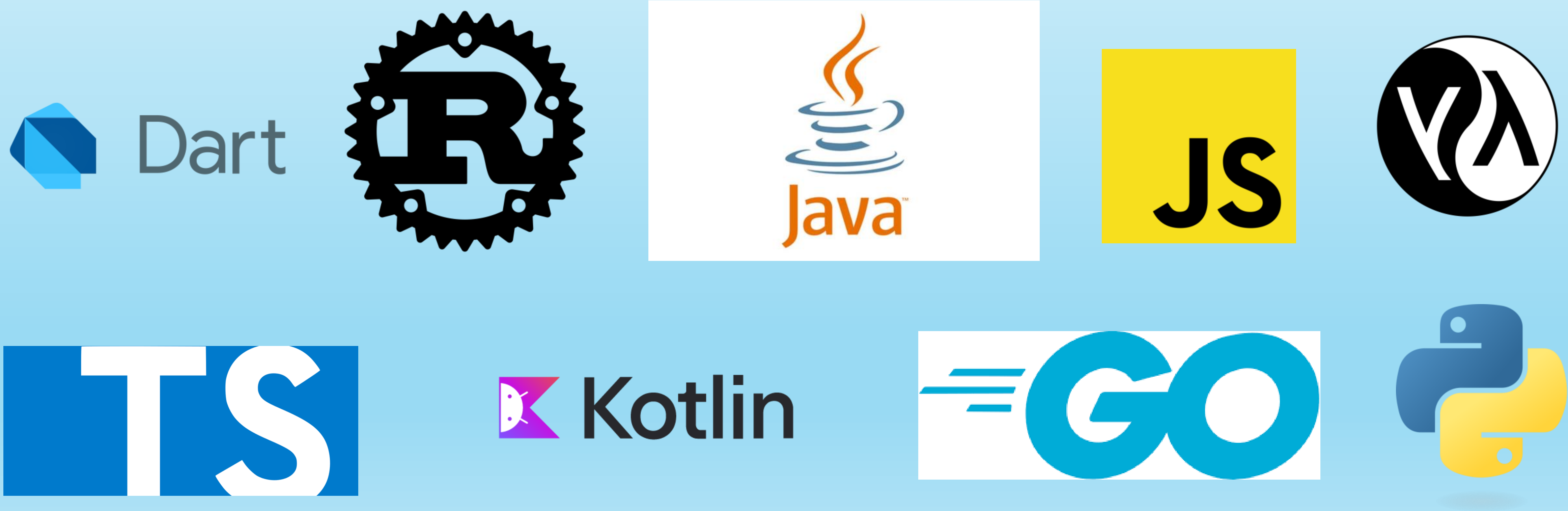


MUTAGENESIS

- Google's implemetation tool.
- Part of the analysis and code review process.



PROGRAMMING LANGUAGES TESTED

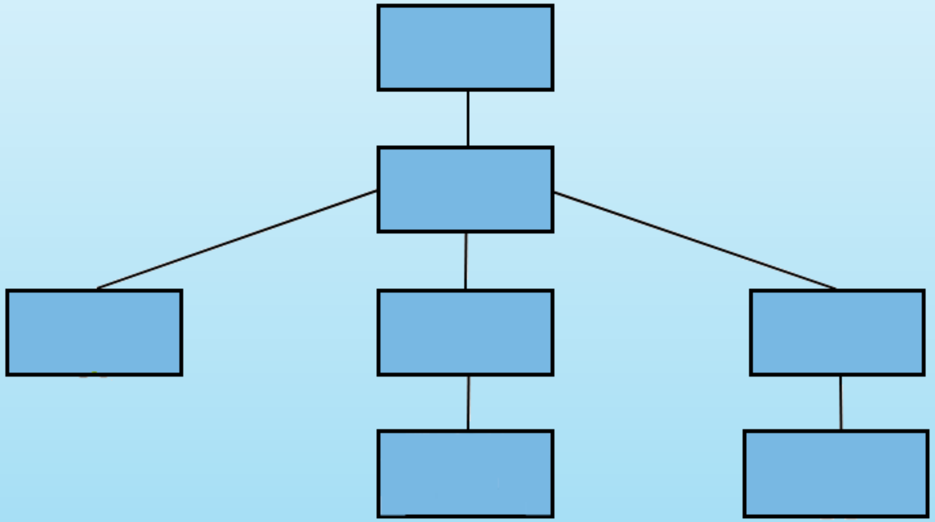


MUTATION TESTING STRATEGY

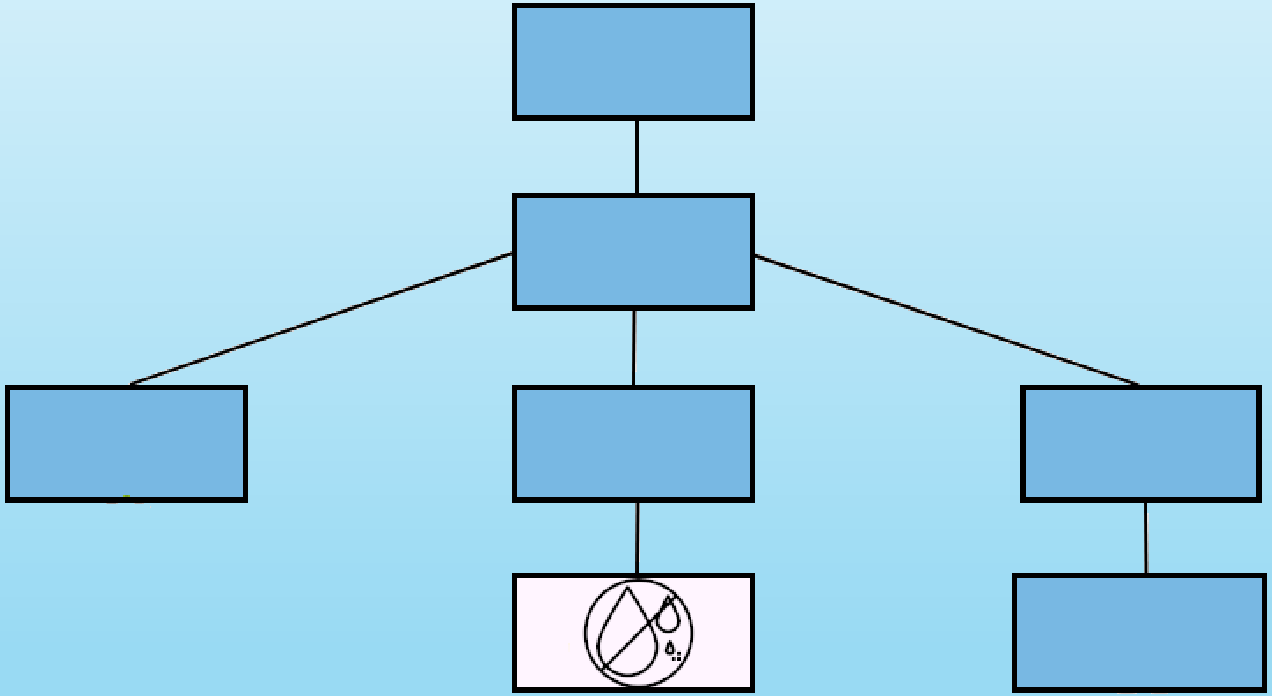
- There is an AST that allows **precise modifications** to source code for mutation testing.
- Mutations apply **only to changed lines in a pull request.**
- Prevents irrelevant changes from distracting developers.

THE ROLE OF AST

- Used to analyze and modify source code for mutations.
- Each language has its **own AST implementation.**
- No universal AST is used due to limitations in **type information and language complexity,** the AST should be adapted to the context.



ARID NODES



SCALING MUTATION TESTING AT

Google

GOOGLE'S IMPLEMENTATION OF MUTATION TESTING

Different from open-source mutation testing approaches because:

- Most of those open-source implementations are usually low level (i.e. bytecode mutation).
- Google's implementation modifies the source code's AST.

This leads to a better visualization for developers of how these mutants work, when compared to low level solutions.

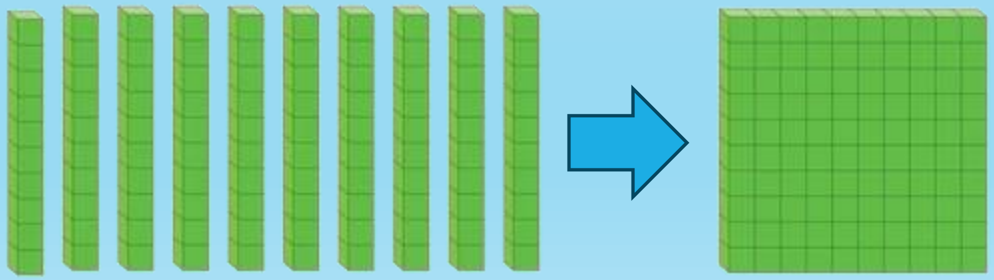
META MUTANTS IN GOOGLE

The result of embedding all of the mutants together is called a ***"meta mutant"*,** which helps achieve scalability.

This approach has not been adopted in Google yet:

- They have a very efficient object caching system, which makes the benefits of this practice a lesser priority.

- In the podcast, Goran voices his interest in trying to put it into use in the future, but he has not had time to get to it for now.



Individual mutants

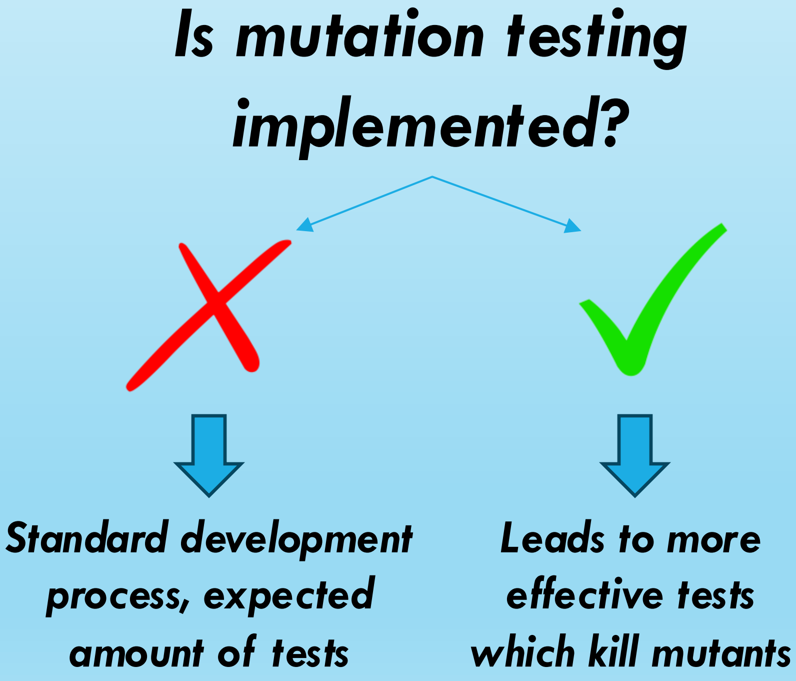
Meta mutant

MUTATION TESTING: EFFECT ON DEVELOPERS

Google ran a study for 6 years, collecting data about millions of mutants.

The results showed that:

- Developers write more tests when mutants come into play, as they are expected to make tests that kill the mutants.
- Said tests actually kill them, and by extension, real bugs too (next slide).



Is mutation testing implemented?

✗ ***Standard development process, expected amount of tests***

✓ ***Leads to more effective tests which kill mutants***

COUPLING EFFECT

Coupling effect/hypothesis: mutants don't necessarily look like real bugs, but tackling them likely leads to bugs getting killed in the process. It's measured by checking how many bugs correspond to a mutant.

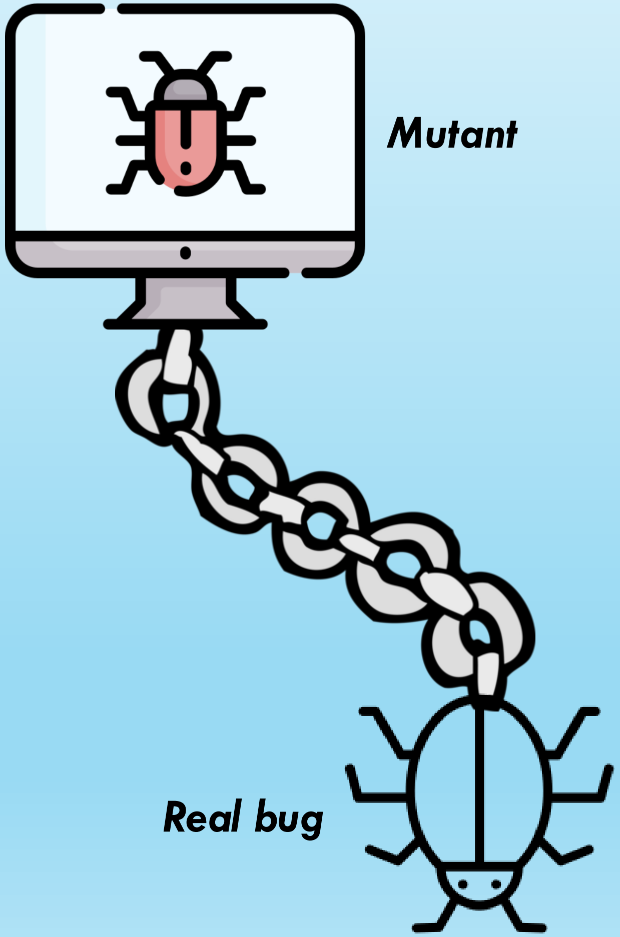
Google conducted an analysis on this:

- Each project operates differently, complicating the process.

- The results obtained showed:

  * In **~70%** of the cases, the bug and the mutant were coupled.

  * The analysis was very expensive, but the results were worth it.



CHALLENGES AND FUTURE OF MUTATION TESTING

Computational Expense

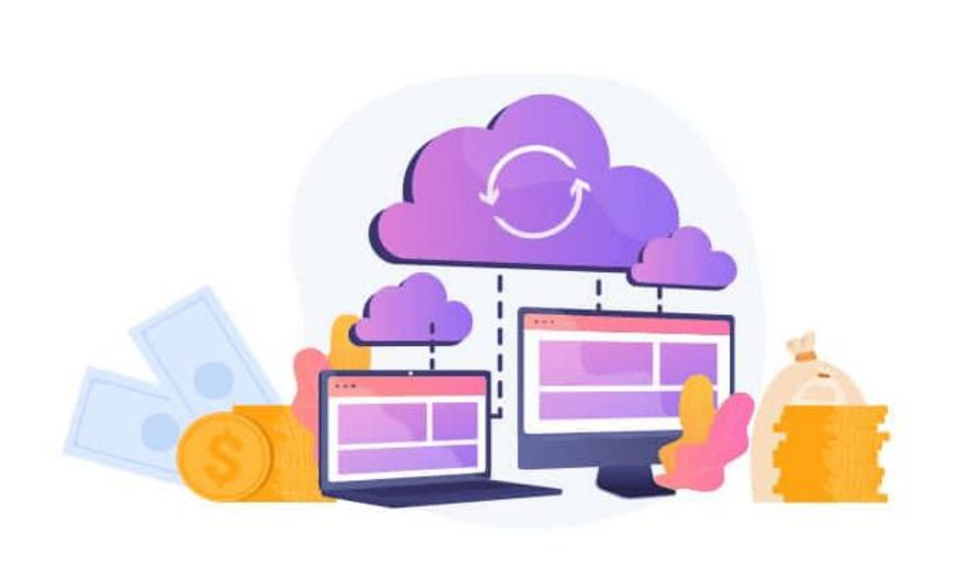
Extremely high number of possible mutations even for small codebases

Creates computational overhead, as each test must be re-executed against every mutant

"Random mutation approaches

proved unsustainable despite being

initially interesting"



CURRENT CHALLENGES

Equivalent Mutants	Mutants that behave identically to original code despite being syntactically different	"It is very difficult to recognize analytically what mutants are equivalent"	Wastes computational resources and human attention
Mutant Quality	"All mutants regarding caching are useless as all of them are equivalent"	Some mutations lead to syntactic errors caught by compilers	Many don't represent realistic programmer errors

FUTURE DIRECTIONS

Intelligent Mutant Selection

Strategic sampling instead of generating all possible mutants

They ended up with 5 or 6 groups of useful mutations

Changes on:

-variables and types

-arrays (e.g. index) and lists

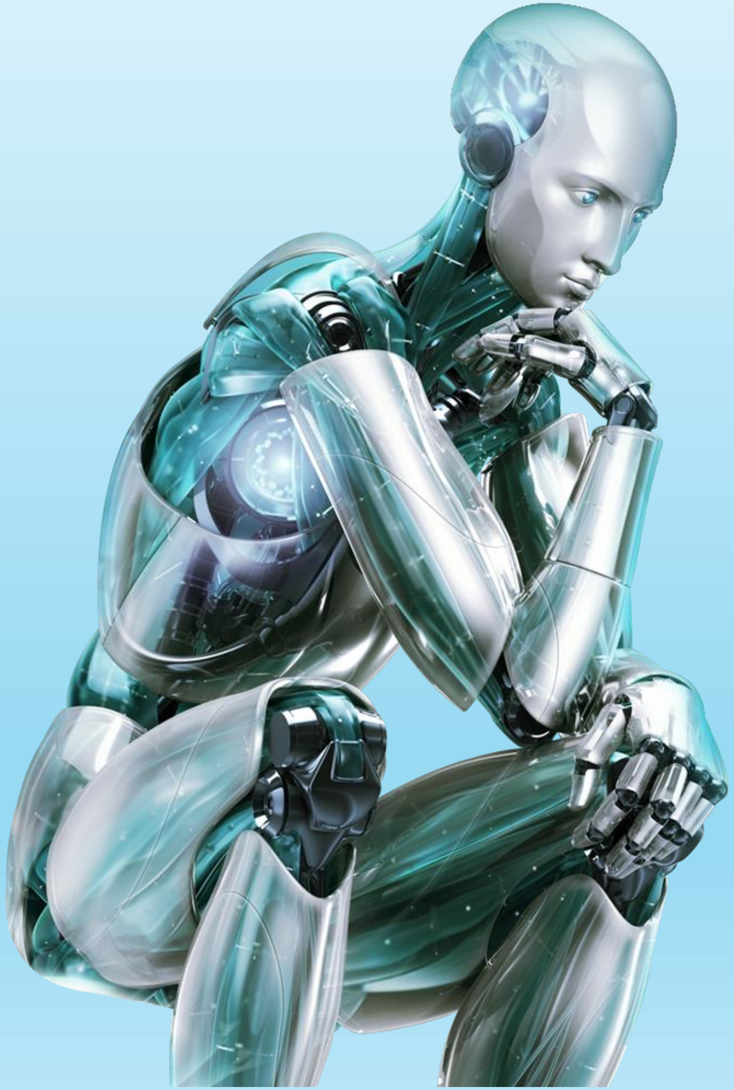
-operators (assignment, arithmetic, logical)

-function/method/service

-modifiers (eg static, transient, synchronized, final, ...)

-inheritance or polymorphism (e.g. casting, super, override, ...)

FUTURE DIRECTIONS



Heuristic Approaches

Search-based software testing using genetic algorithms

"Many improvements can be done with heuristics to discard useless mutants"

Techniques that lead to discover test suites with good testing values

Integration with AI

Tools like TestSpark combining "LLM-based test generation"

More targeted and efficient mutation generation

FUTURE DIRECTIONS

Quality Measurement

"We don't know what code quality is, we cannot measure it!"

But mutation testing helps improve it in practice

Conclusion:

The future lies in making mutation testing more efficient and effective

The goal is improving the overall product, not just killing mutants