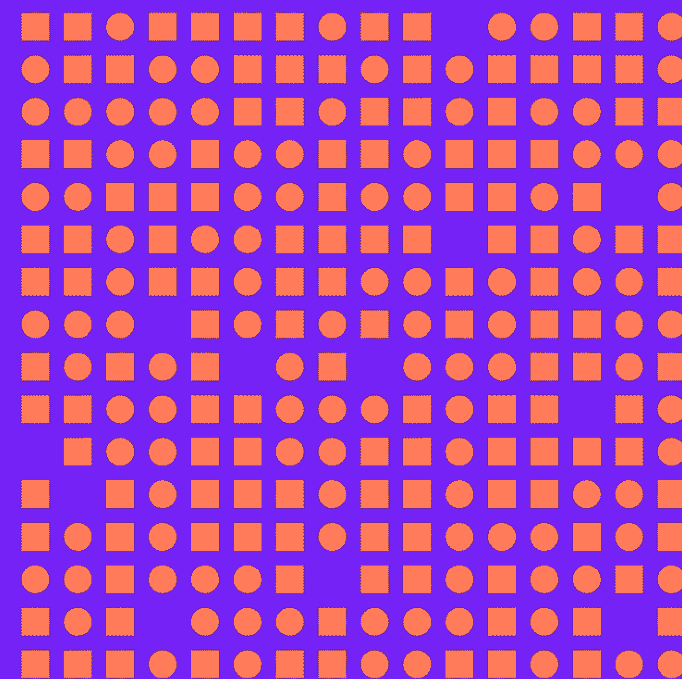


Lanzer

Agent-Driven Fuzzer & Sample Generator
for Langium DSLs

LangDev 2026 · Soulaymen Chouri
soulaymen.chouri@typefox.io



You built a DSL, congratz!

Now you need tests. A LOT of tests.

01

Many types and rules

Every type, rule and reference can combine with the others.
The space of programs is huge.

02

End-to-end, at scale

Hundreds of real programs that parse, type-check and run.

03

Good & bad cases

Valid programs that work, and invalid ones that must fail with the right error.

I.e We want a large corpus.

Writing them by hand doesn't scale.



Two extremes.

Both miss

FAILURE MODE A

Pure LLM

"> Generate tests, don't make mistakes."

- Dummy tests with no specific target
- Doesn't know your grammar
- No structural guarantees about the output
- Goes off track: nonsense code, invented APIs
- Heavy prompt engineering required

Lots of code, no clear target.

FAILURE MODE B

Pure fuzzing

"Generate every AST combinations you can."

- Combinatorial AST generation from the grammar
- Parses, but often fails linking: unknown imports, unresolved names
- Weird names, code nobody would write
- Rarely reaches the type checker, never tests behaviour

Programs that parse, have no meaning.



Meet Lox.

Our test subject for today.

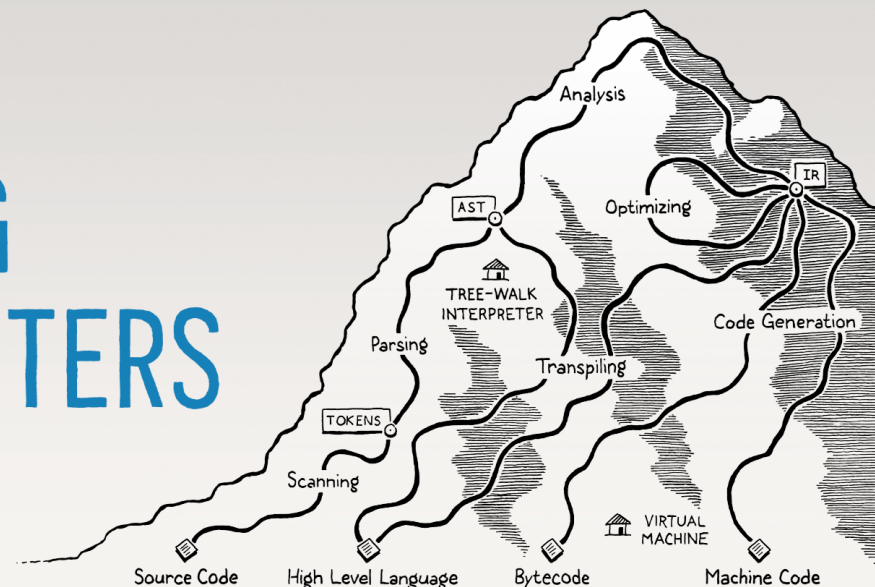
WHERE IT COMES FROM

Lox is the small teaching language from Robert Nystrom's book. **Langium Lox** is our version, with static types.

```
fun factorial(n: number): number {  
  if (n ≤ 1) { return 1; }  
  return n * factorial(n - 1);  
}  
print factorial(5); // 120
```

CRAFTING INTERPRETERS

ROBERT NYSTROM



Now we want tests for it. Lots of them.



The Lox grammar.

Quick sneak peek

```
lox.langium
entry LoxProgram:
  elements+=LoxElement*;

FunctionDeclaration:
  'fun' name=ID '(' parameters... ')'
  ':' returnType=TypeReference body=ExpressionBlock;

FeatureCall infers Expression:
  {infer MemberCall}
  element=[NamedElement:ID] ('(' arguments... ')')?;

VariableDeclaration returns NamedElement:
  {infer VariableDeclaration} 'var' name=ID
  (':' type=TypeReference)? (assignment?='=' value=Expression)?;

type NamedElement = Parameter | FunctionDeclaration
  | VariableDeclaration | MethodMember | FieldMember | Class;
```

LoxProgram

The entry rule

FunctionDeclaration

A typed function: name, parameters, return type, body.

MemberCall → cross-reference

A call. **element** points to what it calls.

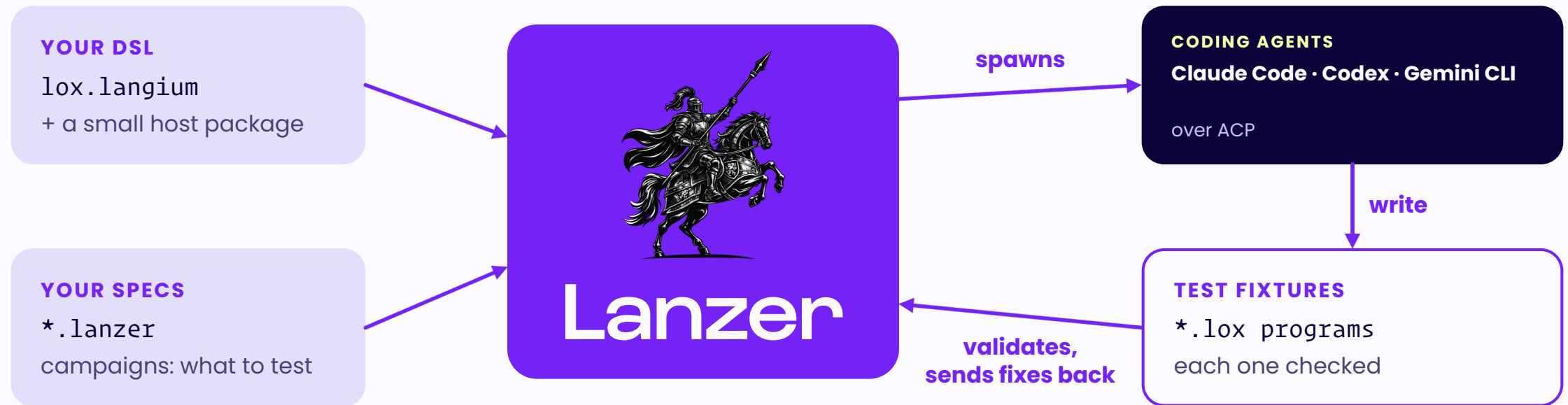
NamedElement

Anything with a name. A call can point to any of them.



Meet Lanzer.

A DSL + tools to write semi-formal specs for test fixtures.



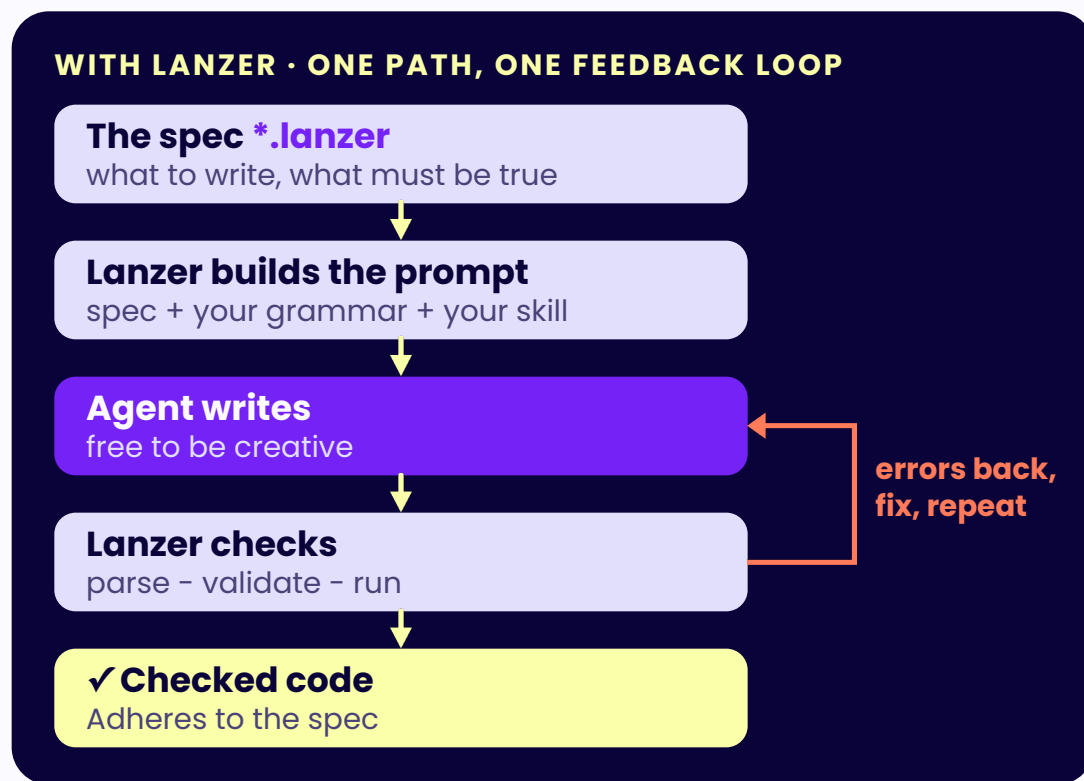
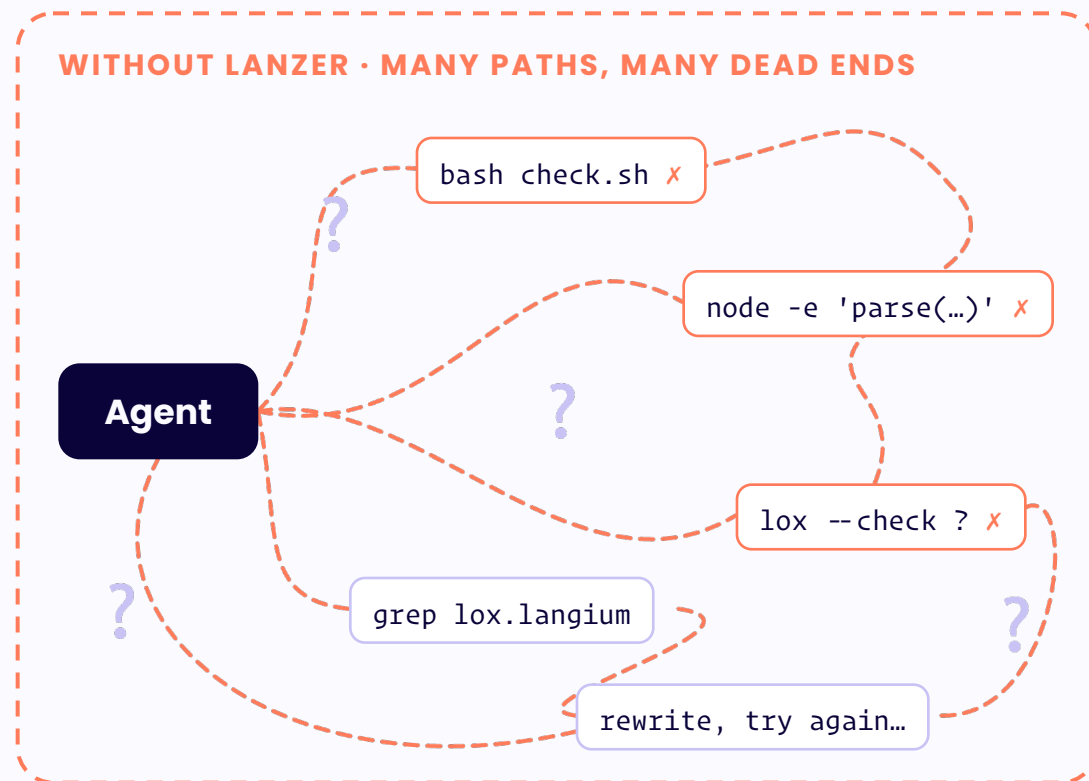
Lanzer does not write the code. It tells agents what to write, checks what they wrote, and sends back what failed.

The goal: lots of code that follows your language. If the agent's fixes stop changing anything, for example because it thinks your validator is wrong, Lanzer ends the fix loop and reports the failure.



Why Lanzer?

Left alone, an agent invents its own workflow, every time.



Rigid workflow, creative code. Lanzer fixes *how* the agent works, not *what* it writes.



Lanzer: A semi-formal DSL.

Formal where structure matters. Informal where intent matters.

FORMAL

What must be there

Lanzer dictates the skeleton: which nodes, which references, how many.

```
require FunctionDeclaration[name="factorial"]
require FunctionDeclaration[name="main"]
  >> MemberCall[element→
    FunctionDeclaration[name="factorial"]]
min 2 FunctionDeclaration
forbid ForStatement
```

INFORMAL

What it is for

Free prose about purpose, behaviour and edge cases. It says what the program should do.

"factorial is recursive. main prints the factorials of 1 to 5, one per line, with a while loop."

The agent fills in the blanks.



A campaign

```
import "lox.langium"

campaign factorial {
  description "Print the factorials of 1 to 5, one per line."
  workspace "test/factorial"
  file main at "src/main.lox" generates LoxProgram {
    require FunctionDeclaration[name="factorial"]
    >> MemberCall[element→
      FunctionDeclaration[name="factorial"]]
  }

  run main {
    expect runs
    expect output "1\n2\n6\n24\n120\n"
    expect not output contains "nil"
  }
}
```

import · your grammar. All types come from it.

workspace · where the agent writes.

file ... generates · a file, and the rule its root must be.

description · the intent, in prose.

require · forbid · min · the structure. Here: factorial must call itself.

run ... expect · what it must do when it runs.



Selectors, borrowed from CSS.

...and typed by your grammar.

TYPE + PROPERTY

```
FunctionDeclaration[name="main"]  
FunctionDeclaration[name^="get"] // starts with  
FunctionDeclaration[returnType] // is set  
= != ^= $= *= and plain presence.
```

COMBINATORS

```
Class > MethodMember // direct child  
FunctionDeclaration >> MemberCall // inside  
Reachability is checked against the grammar.
```

CROSS-REFERENCES

```
MemberCall[element →  
    FunctionDeclaration[name="factorial"]]  
→ follows a Langium cross-reference: a call that resolves to  
factorial.
```

PSEUDO-CLASSES

```
Class:has(MethodMember[name="init"])  
Class:not(MethodMember[name="init"])  
:has / :not · contains, or doesn't contain.
```



Impossible asks are rejected **before any token is spent.**

```
bad.lanzer
```

```
1 import "lox.langium"  
2  
3 campaign bad {  
4   workspace "out"  
5   file f at "main.lox" generates LoxProgram {  
6     require FunctionDeclaration[nme="factorial"]  
7     require Class > PrintStatement  
8     require Lambda  
9   }  
10 }
```

```
$ npx lox-lanzer validate bad.lanzer
```

```
6:37 Type 'FunctionDeclaration' has no property  
'nme'. Did you mean 'name'?
```

```
7:25 'PrintStatement' is not reachable as a direct  
child of 'Class' in the imported grammar. It is a  
descendant, though, via 'ClassMember' >  
'ExpressionBlock': use '>>'.
```

```
8:17 Could not resolve reference to AbstractRule  
named 'Lambda'.
```

No agent started, zero tokens spent. The same errors show up as you type, in VS Code.



Plug in your DSL

A new package that overrides some services

```
class LoxLanzerService extends DefaultLanzerService {  
  // advice for the prompt, on top of the grammar  
  getGenerationPolicy(job) { ... }  
  
  // where your agent skill lives  
  dslSkill() { return { name: 'write-lox', path }; }  
  
  // runs a program, for `run` blocks  
  execute(request) { ... }  
  
  // every code, so `expect error` is checked early  
  diagnosticCodes() { return ['LOX_TYPE_NOT_ASSIGNABLE', ...]; }  
}
```

`lox-lanzer-service.ts`

What the agent is told, how code runs

`lox-host.ts`

Wiring: Lox on Lanzer's services

`cli.ts`

All commands, from createLanzerHostCli

`bin/lox-lanzer.js`

The entry point

Template: `packages/lanzer-lox`



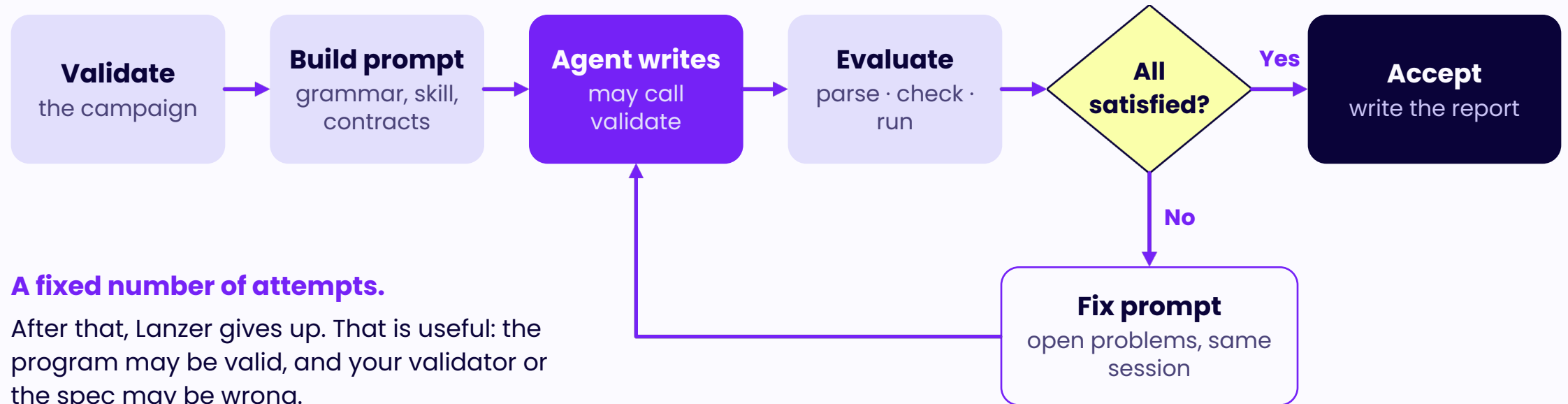
ACP 101.

One vendor-neutral protocol for coding agents.



How a run works

The Lanzer workflow.



A fixed number of attempts.

After that, Lanzer gives up. That is useful: the program may be valid, and your validator or the spec may be wrong.

If it fails, the report says where. For example: a syntax error, a missing function, or the wrong output.



You write the spec.

Lanzer transforms to prompt

YOUR SPEC · factorial.lanzer · 14 LINES

```
import "lox.langium"

campaign factorialLox {
  description "Print the factorials of 1 through 5, one per line,
              computed by a recursive function."
  workspace "test/factorial"

  file mainFile at "src/main.lox" generates LoxProgram {
    require FunctionDeclaration[name="factorial"]
    require FunctionDeclaration[name="factorial"]
    >> MemberCall[element→FunctionDeclaration[name="factorial"]]
  }

  run mainFile {
    expect runs
    expect output "1\n2\n6\n24\n120\n"
    expect not output contains "nil"
  }
}
```

THE PROMPT LANZER BUILDS · 67 LINES, SHORTENED

Campaign: factorialLox
Campaign description: Print the factorials of 1 through 5, one per line, ...
Workspace root: ../examples/test/factorial
Read the grammar reference from this absolute path before generating: ../lox-1785785025.bnf
Use the installed agent skill named "write-lox" before generating.
You must write the following generated files in this run:
- mainFile: ../test/factorial/src/main.lox [LoxProgram]
Tools available to you for this task:
- `mcp\_\_lanzer\_\_grammar\_reference` returns the target language's full grammar. ...
- `mcp\_\_lanzer\_\_validate` ... runs the exact check this task is graded by ...
Execution instructions:
- Generate the full declared file set for campaign factorialLox. ...
- This is the statically-typed `langium-lox` dialect, NOT the dynamically-typed Lox ...
Required practices: - Every variable, parameter, and function is statically typed. ...
Forbidden practices: - Do NOT use `%` (modulo) or any exponentiation operator. ...
Host reference files available for reading: write-lox-example: ../examples.lox
Selector syntax used in the requirements below: - `A >> B` - B must appear ... inside A. ...
Per-file generation targets:
- mainFile → ../src/main.lox [LoxProgram]
- MUST contain FunctionDeclaration[name="factorial"].
- MUST contain FunctionDeclaration[name="factorial"] >> MemberCall[element→...].
Behaviour checks - once the files are valid, Lanzer runs each program below ...
- MUST run to completion, without a runtime error or timeout.
- output MUST be exactly "1\n2\n6\n24\n120\n" ...
- output MUST NOT contain "nil".
Produce this behaviour by computing it, ... not by printing the expected text directly.

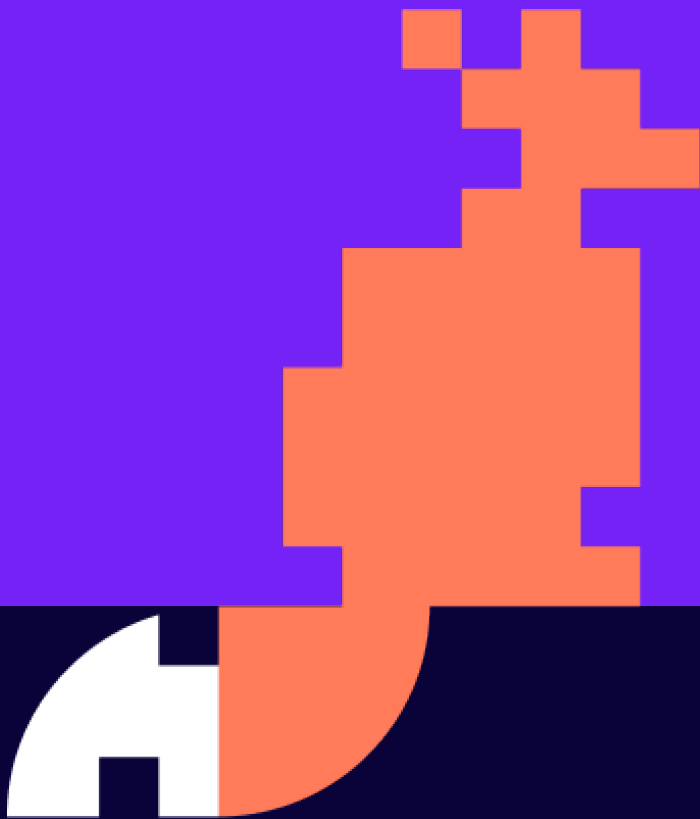
■ your campaign ■ generated from your .langium ■ your host · skill, policy ■ Lanzer · tools, rules



TypeFox

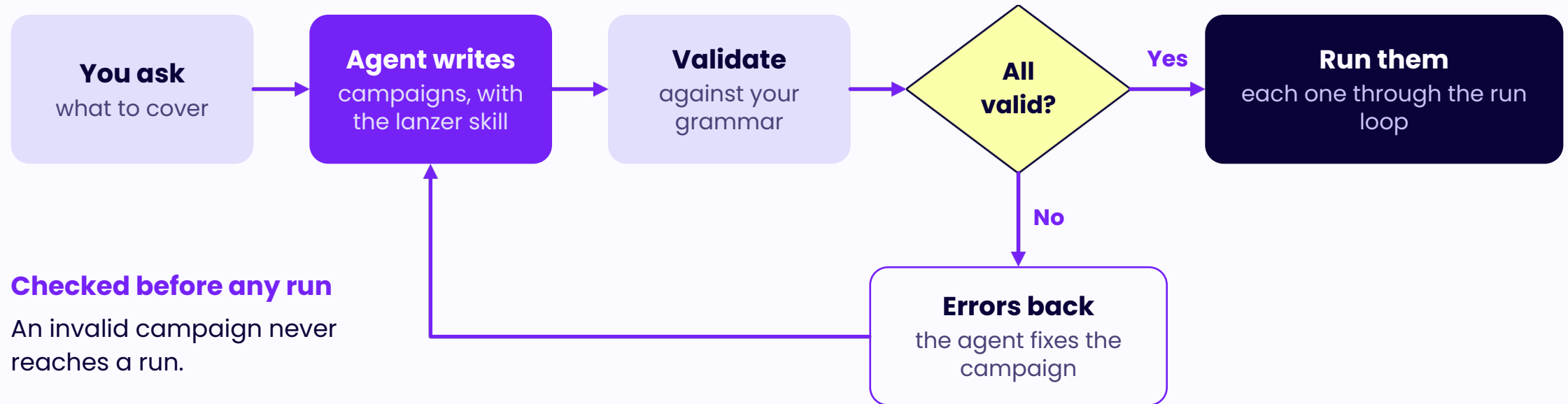
Demo #1

One campaign



Generate campaigns

Same loop, one level up: a coding agent writes the fixtures.



```
> Using the lanzer skill, write 10 Lox campaigns in ./campaigns covering closures,  
classes and type errors. Validate each one, then run npx lox-lanzer generate ./campaigns
```



Compare agent skills

```
$ npx lox-lanzer generate ./bench --runs 3 --isolated --no-skill --report noskill.json
$ npx lox-lanzer generate ./bench --runs 3 --isolated --report skill.json
$ npx lox-lanzer compare noskill.json skill.json
```

	a	b	change
pass rate	8/15 (53%)	13/15 (87%)	+33 pts
cost per run	0.1775 USD	0.1479 USD	-17%
time per run	92.9s	59.6s	-36%
tokens per run	144,790	155,754	+8%

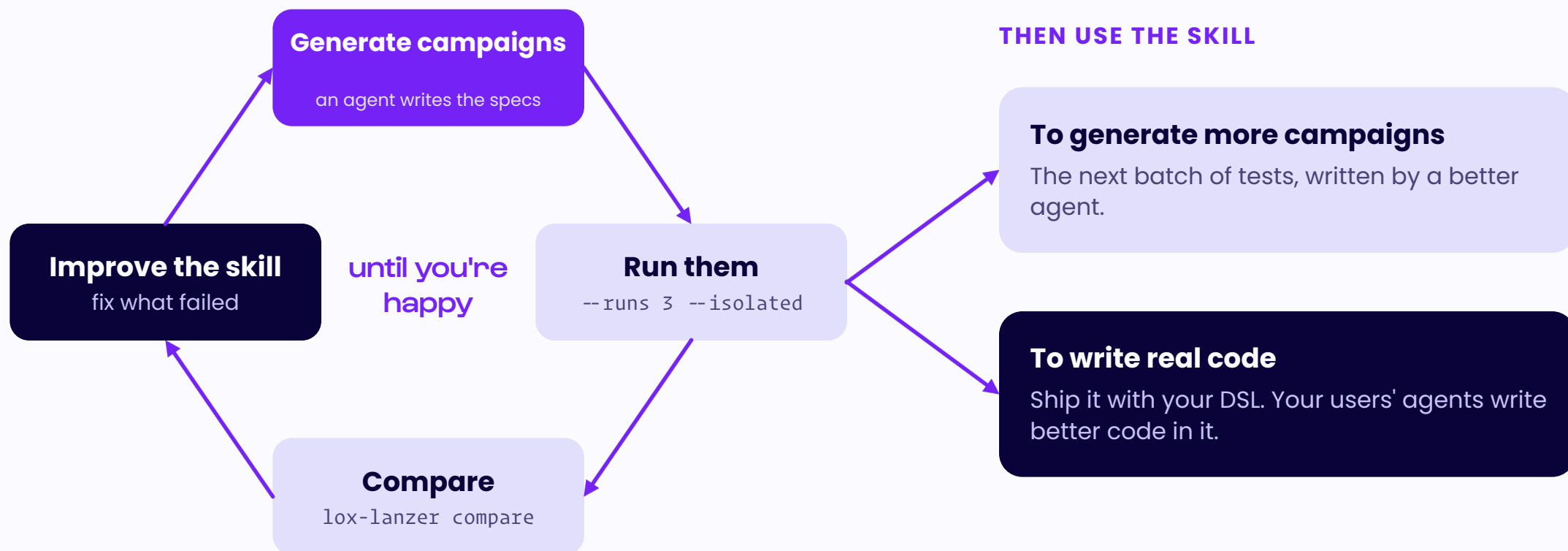
by campaign:

campaign	a	b	change
higherOrder	3/3	3/3	unchanged ±0 pts
integerMath	0/3 (2× semantics, 1× behaviour)	1/3 (2× semantics)	improved +33 pts
linkedStack	2/3 (1× behaviour)	3/3	improved +33 pts
negativeSubclass	3/3	3/3	unchanged ±0 pts
shapes	0/3 (1× semantics, 2× behaviour)	3/3	improved +100 pts



Improve your agent skill, in a loop.

Generate, run, compare, improve. Repeat.



One more thing...



A spec for writing a spec

Lanzer is a Langium language too, so it can test itself.

```
meta/lox_call_on_non_function.lanzer · 1 of 12, one per Lox error code

import "lanzer.langium"

campaign cover_lox_call_on_non_function {
  description "Write ONE campaign that tests whether Lox reports LOX_CALL_ON_NON_FUNCTION:
    a negative file wrong in exactly one way, and a clean twin."

  file spec at "call.lanzer" generates CampaignFile {
    require Campaign > FileSpec:has(DiagnosticExpectation[code="LOX_CALL_ON_NON_FUNCTION"])
    require Campaign > FileSpec:not(DiagnosticExpectation) // a clean twin
    require FileSpec[description] // intent written down
    forbid RunSpec // no guessed outputs
    require SelectorPart[rule→AbstractRule[name="MemberCall"]] // must test Lox calls
  }
}
```

generates CampaignFile

Its output is itself a campaign.

rule→AbstractRule[...]

The last line reaches into Lox's grammar: the campaign must be about calls.



An agent writes the campaign

A negative file, and its clean twin.

```
lox_call_on_non_function_1.lanzer
import "lox.langium"

campaign callOnNonFunction {
  file badCallFile at "bad_call.lox" generates LoxProgram {
    description "Declares a number 'count', then calls it: count()."
    require VariableDeclaration[name="count"]
    require FunctionDeclaration[name="main"] >> MemberCall[explicitOperationCall]
    expect error code "LOX_CALL_ON_NON_FUNCTION"
  }

  file goodCallFile at "good_call.lox" generates LoxProgram {
    description "Clean twin: the same 'count' and main(), but main prints it instead of calling it."
    require VariableDeclaration[name="count"]
    require FunctionDeclaration[name="main"] >> PrintStatement
  }
}
```

✓ valid

✓ meets its meta
spec

This spec ran twice: **2**
campaigns, both
valid.

Across all 12 specs: 24 of
24 valid, \$3.50.



Agents write the programs

Five agents, one campaign.

bad_call.lox · runs 2, 3, 5

X Lox reports nothing

```
var count: number = 5;
fun main(): void {
  count();
}
```

good_call.lox · the twin, all 5 runs

✓ no errors

```
var count: number = 5;
fun main(): void {
  print count;
}
```

bad_call.lox · run 1

X Lox reports nothing

```
var count = nil;
fun main(): void {
  count();
}
```

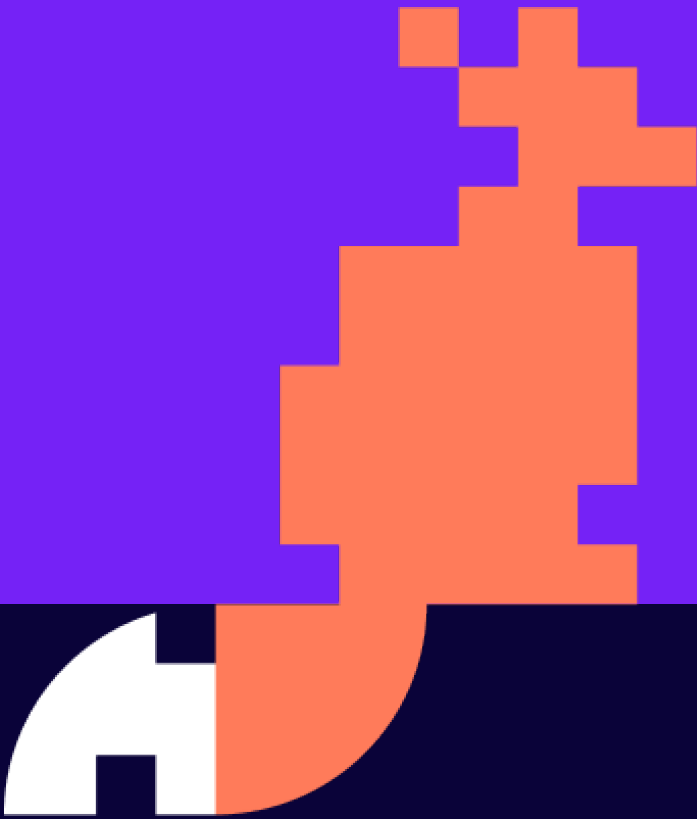
FILE, 5 RUNS	EXPECTED	LOX REPORTED
bad_call.lox	LOX_CALL_ON_NON_FUNCTION	nothing, 5 of 5
good_call.lox	no errors	no errors



TypeFox

Demo #2

Lanzer tests Lanzer.



TypeFox

Thank you

Questions?

github.com/TypeFox/lanzer

typefox.dev/lanzer

