Фаззинг как основа эффективной разработки на примере LuaJIT

Сергей Бронников

Разработчик в Tarantool

О чём доклад?

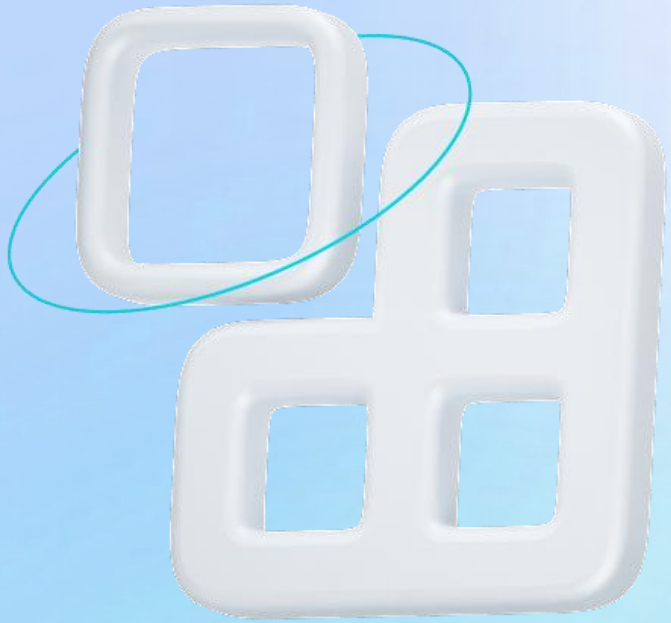
- Про эффективную разработку сложного программного компонента
- Про использование фаззинга в процессе разработки ПО
- Про социальные аспекты разработки, которые можно иногда решить технически

Ранее:

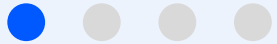
О чём я говорю, когда говорю о тестировании корректности работы компилятора (HighLoad 2022)

План доклада

- Знакомство с языком Lua и LuaJIT
- Внутреннее устройство LuaJIT
- Рассказ про LuaJIT в Tarantool
- Использование фаззинга в процессе разработки LuaJIT в Tarantool



01
Знакомство с Lua и LuaJIT

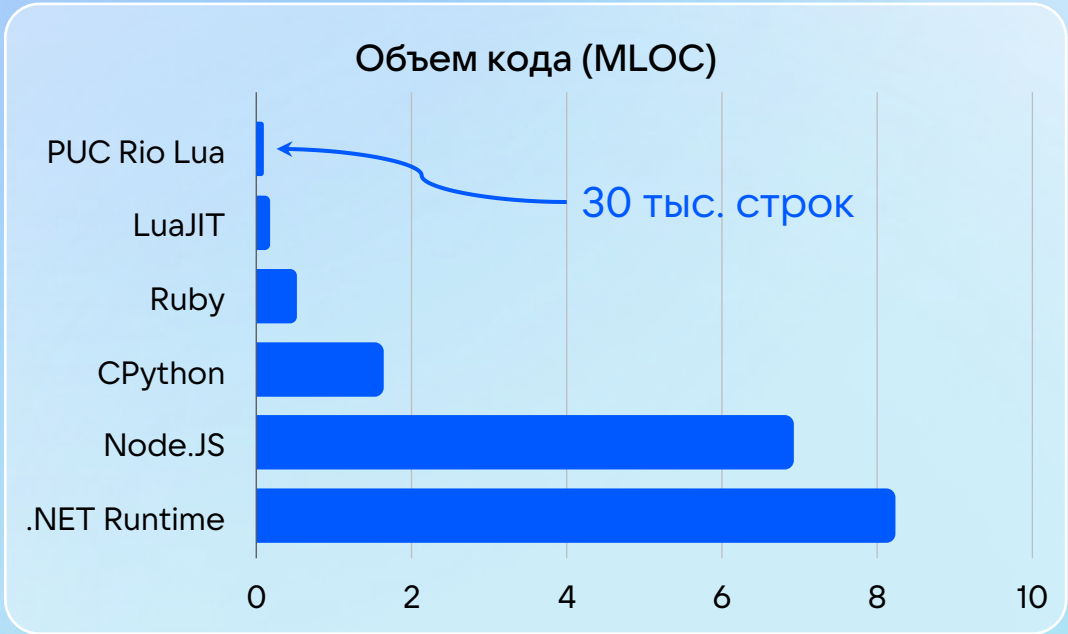


Язык Lua: Знакомство

- Легковесный мультипарадигменный язык программирования
- Спроектирован простым и легко встраиваемым
- Нет спецификации языка, основной документ – Lua Reference Manual
- Изредка используется как основной язык в проекте: Prosody, Kong
- Широко используется для программирования внутренней логики приложений: OpenResty, HAProxy, Redis, Roblox, Wireshark, NeoVim, mpv, VLC, Snort, Pandoc, PowerDNS, Snort, NMap, LÖVE, Darktable, WoW, Angry Birds, Adobe Lightroom, Torch, Tarantool

PUC Rio Lua: Знакомство

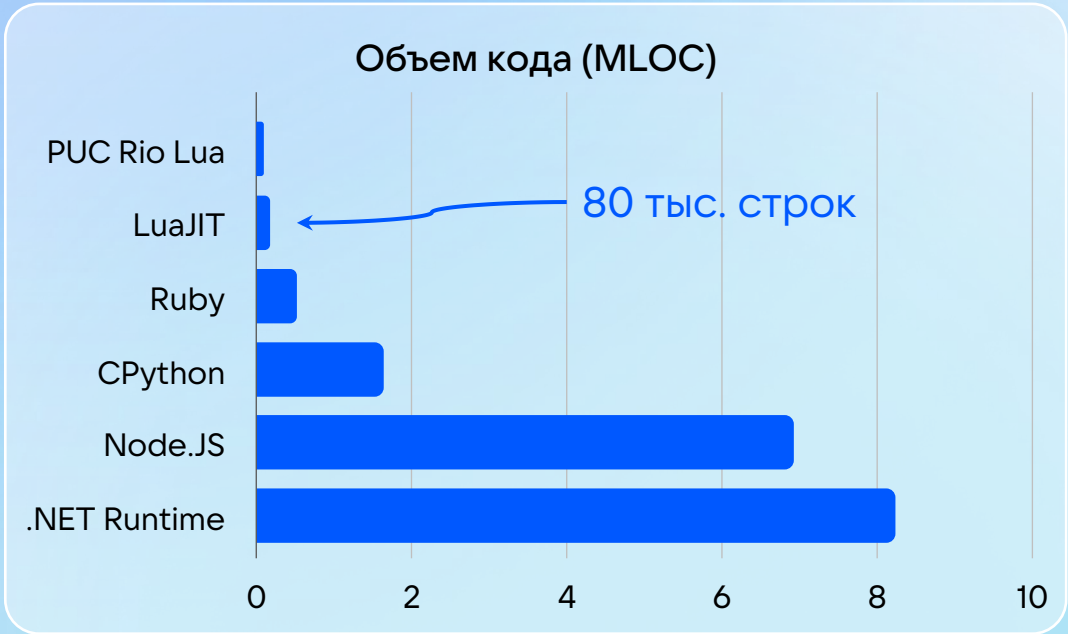
- PUC Rio Lua – референсная реализация языка
- Разрабатывается с 1993 года группой из трех разработчиков университета Рио-де-Жанейро
- Разработка закрытая, исходный код под лицензией MIT
- Написан на языке Си (C99)



LuaJIT: Знакомство

- Одна из наиболее производительных реализаций динамического языка программирования
- Быстрый интерпретатор + JIT
- Проект публично разрабатывается с 2005 г.
- Реализует Lua 5.1 с некоторой примесью Lua 5.2+
- Мультиплатформенный: x86/x64, ARM, ARM64, MIPS, PPC
- Закрытая разработка, исходный код под лицензией MIT

LuaJIT



LuaJIT: Аспекты проекта

Социальные

- Проект единолично разрабатывает Mike Pall
- Общение только через тикеты
- Тикеты без стабильного минимизированного репродьюсера Майк закрывает
- За споры с ментейнером в тикетнице можно получить бан
- Разработка закрытая – изменения в проекте становятся известными по факту

Технические

- Нет тестов (фикс фикса для предыдущего фикса)
- Отсутствие документации
- Отсутствие понятного кода
- Плохая предсказуемость производительности

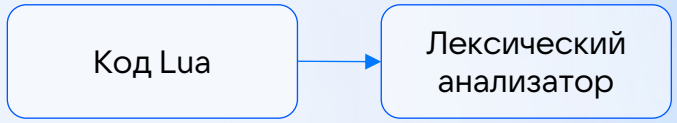
Как следствие появлялись форки LuaJIT: Tarantool, OpenResty, LuaVela, RaptorJIT. Каждый форк живет своей собственной жизнью, сотрудничества нет

02
Устройство LuaJIT

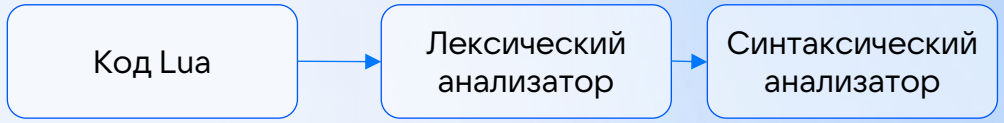
LuaJIT: Внутреннее устройство

Код Lua

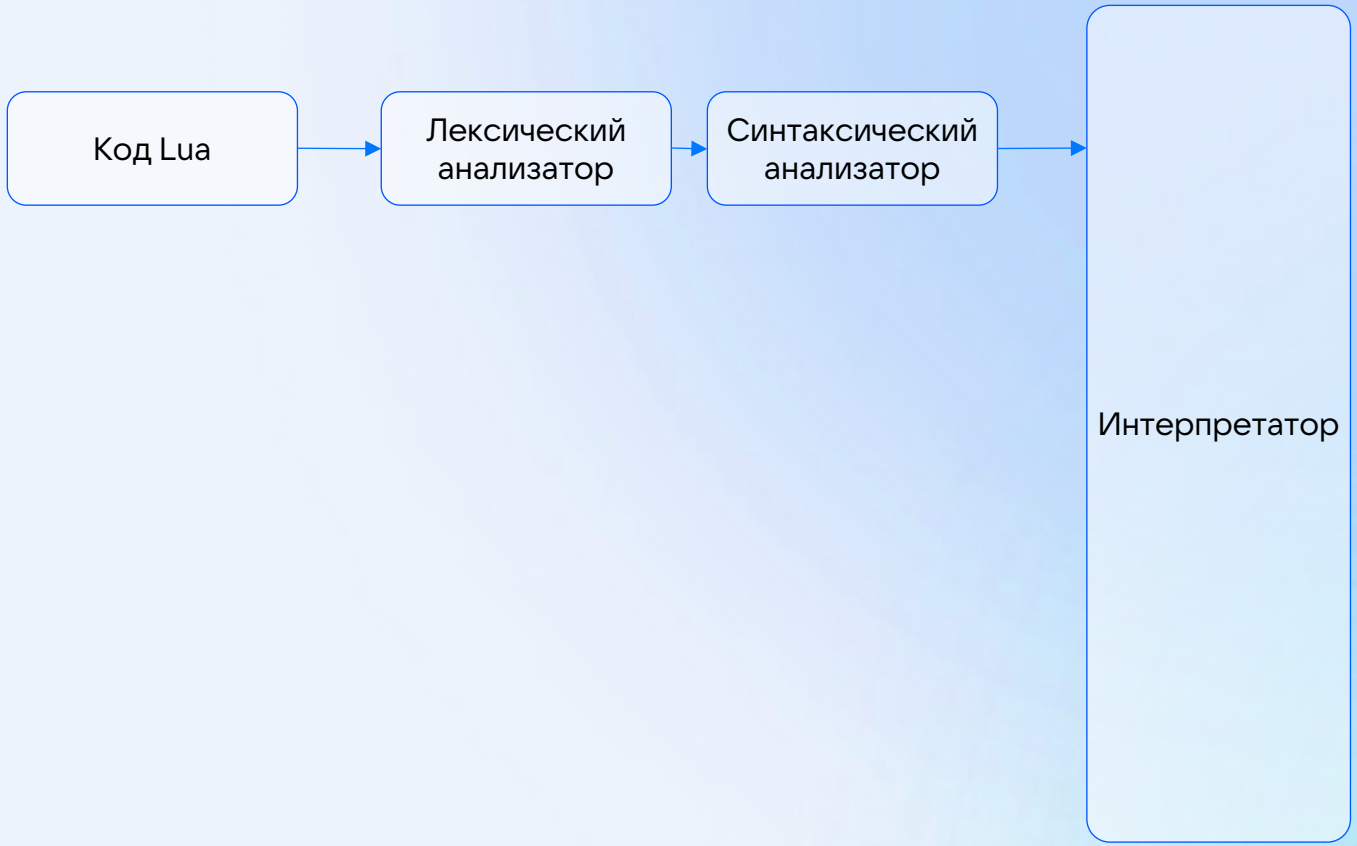
LuaJIT: Внутреннее устройство



LuaJIT: Внутреннее устройство



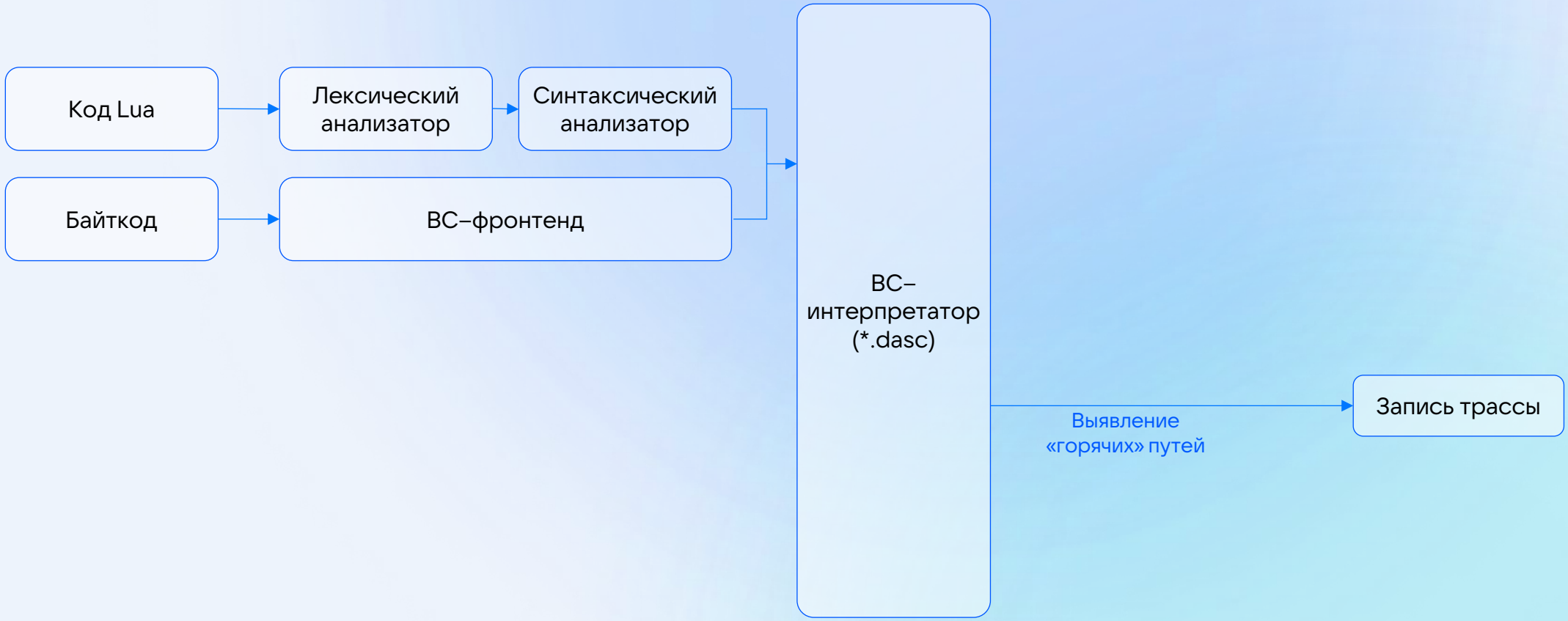
LuaJIT: Внутреннее устройство



tech

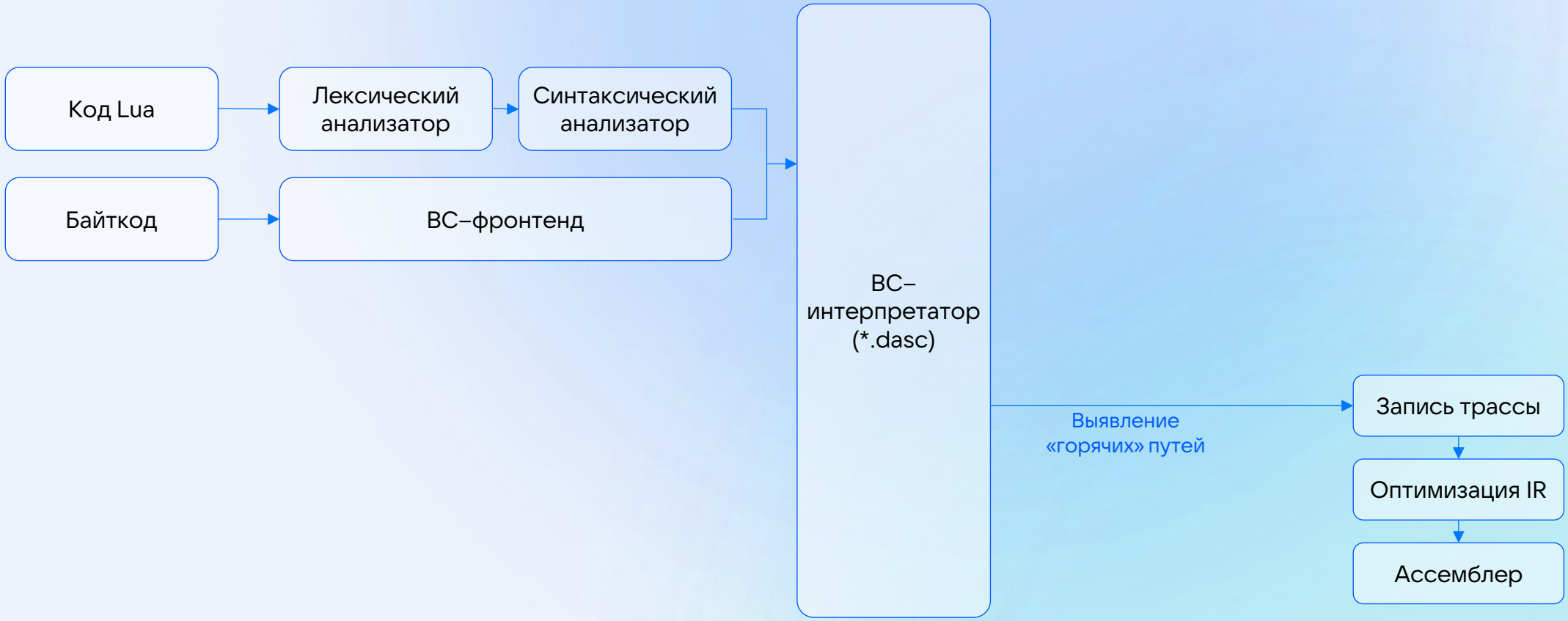
LuaJIT: Внутреннее устройство

LuaJIT: Внутреннее устройство

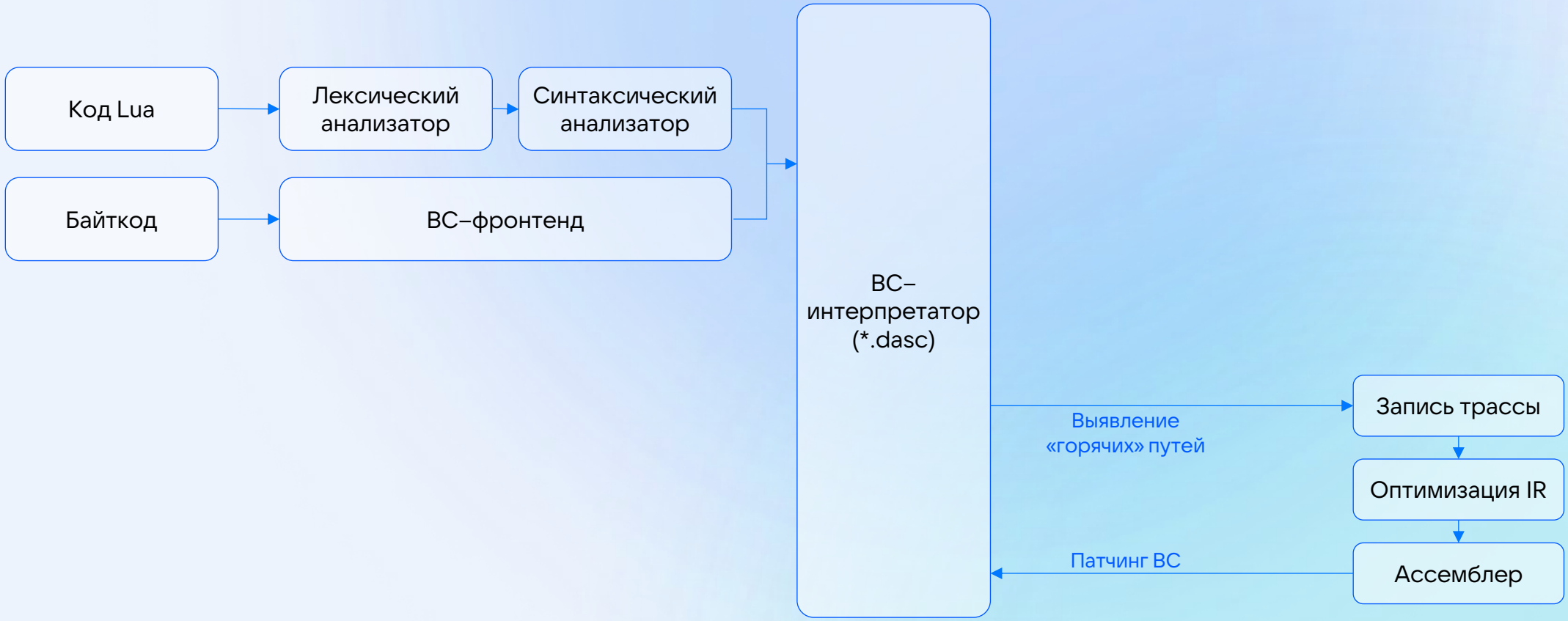


tech

LuaJIT: Внутреннее устройство

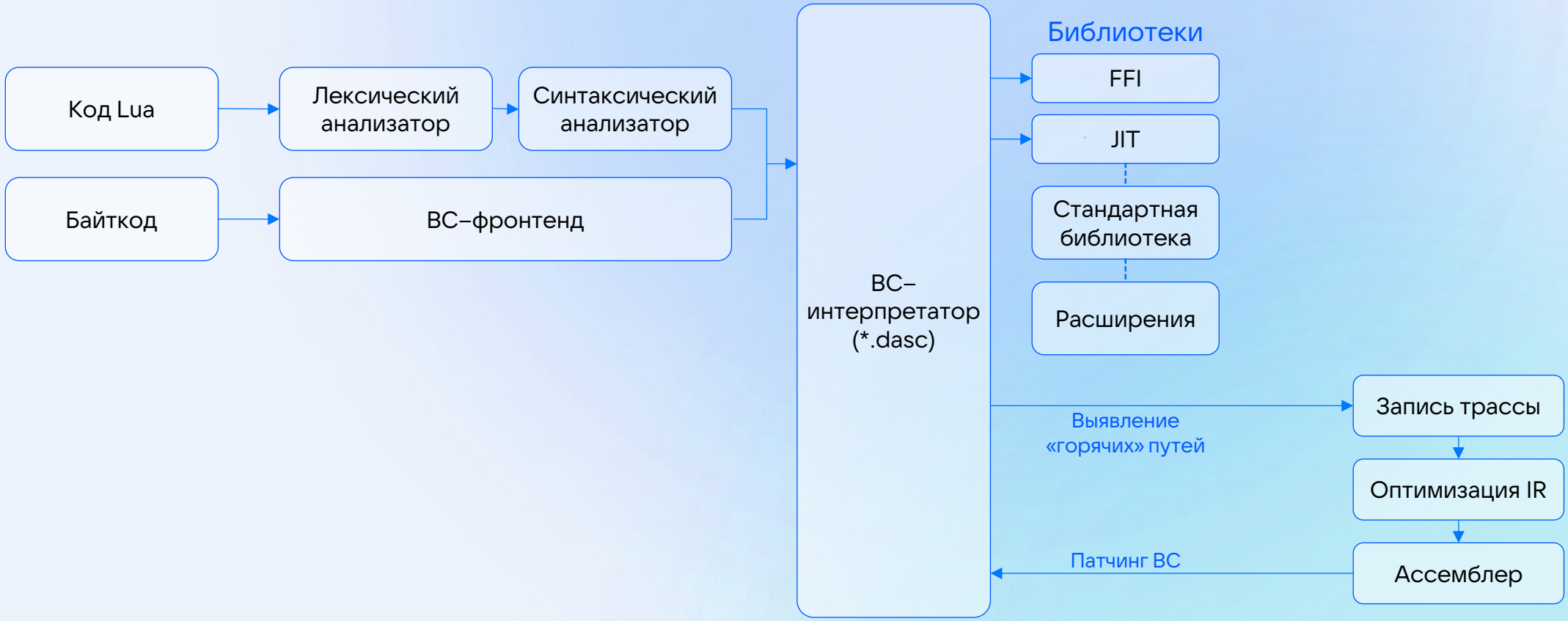


LuaJIT: Внутреннее устройство

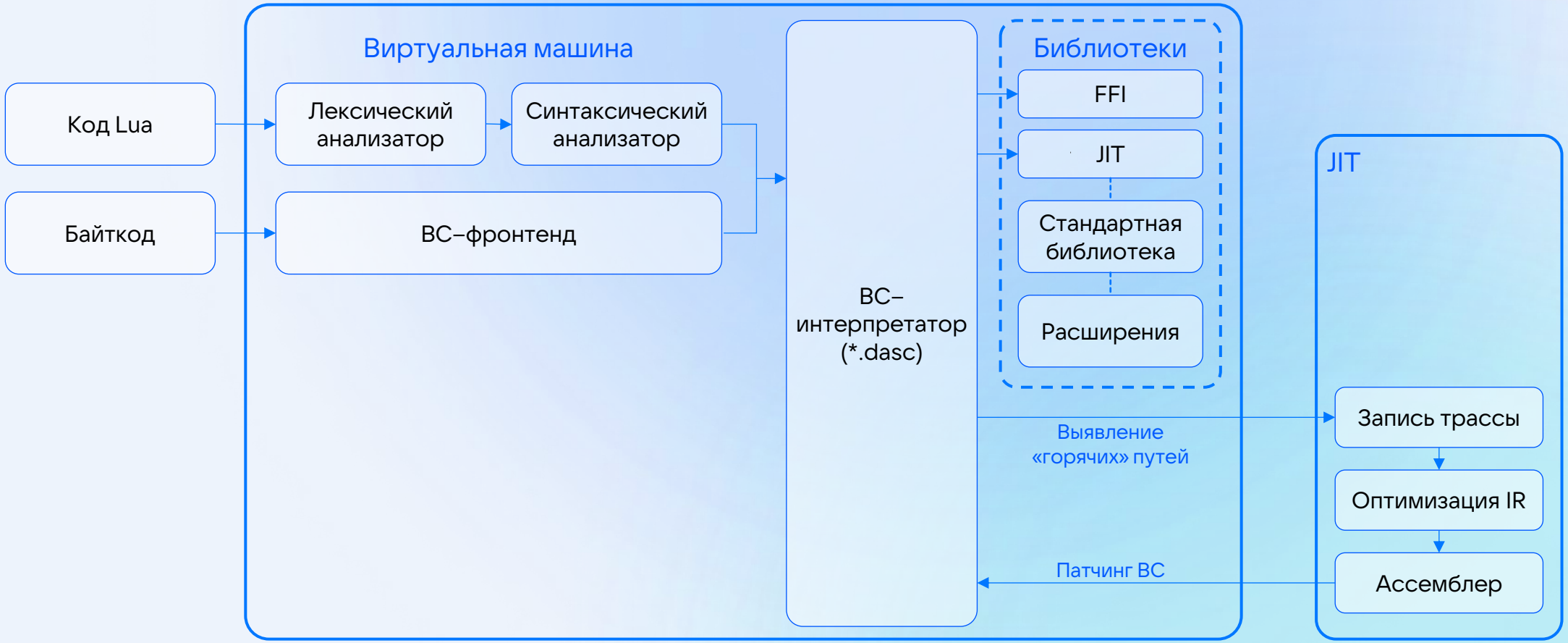


tech

LuaJIT: Внутреннее устройство

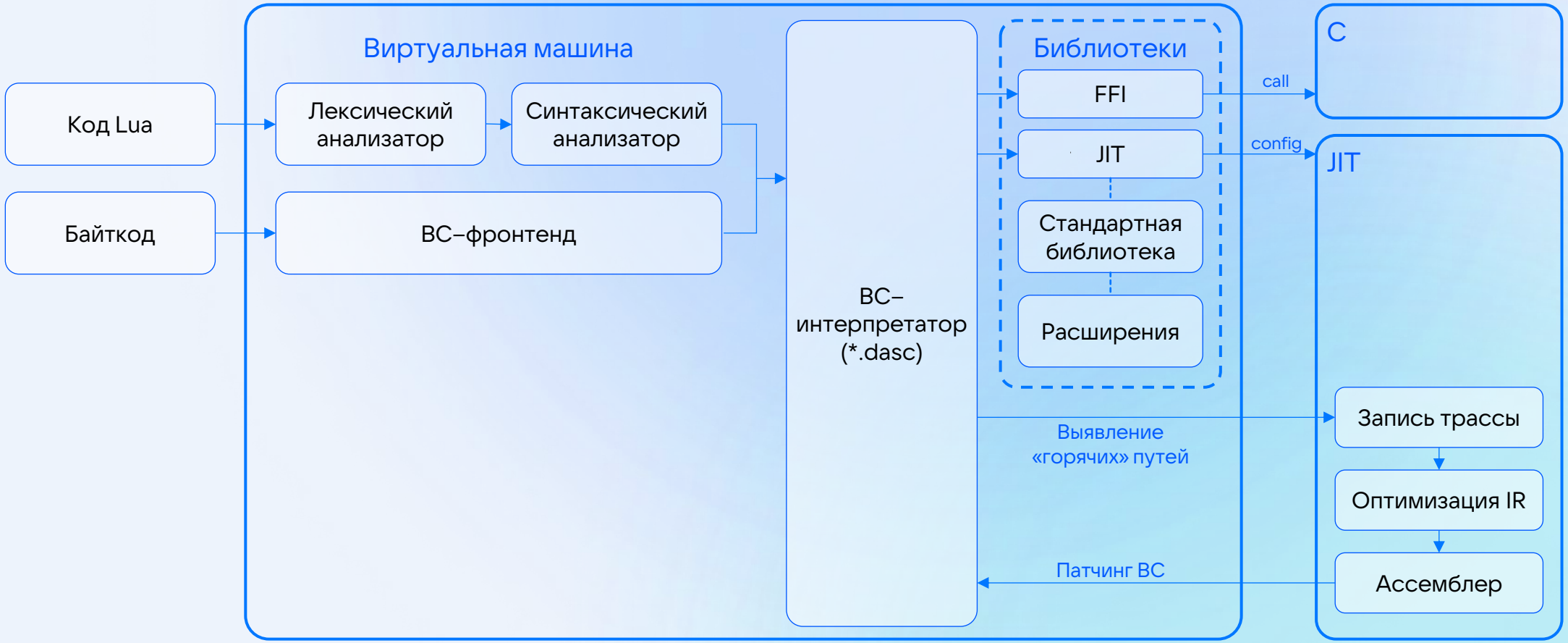


LuaJIT: Внутреннее устройство



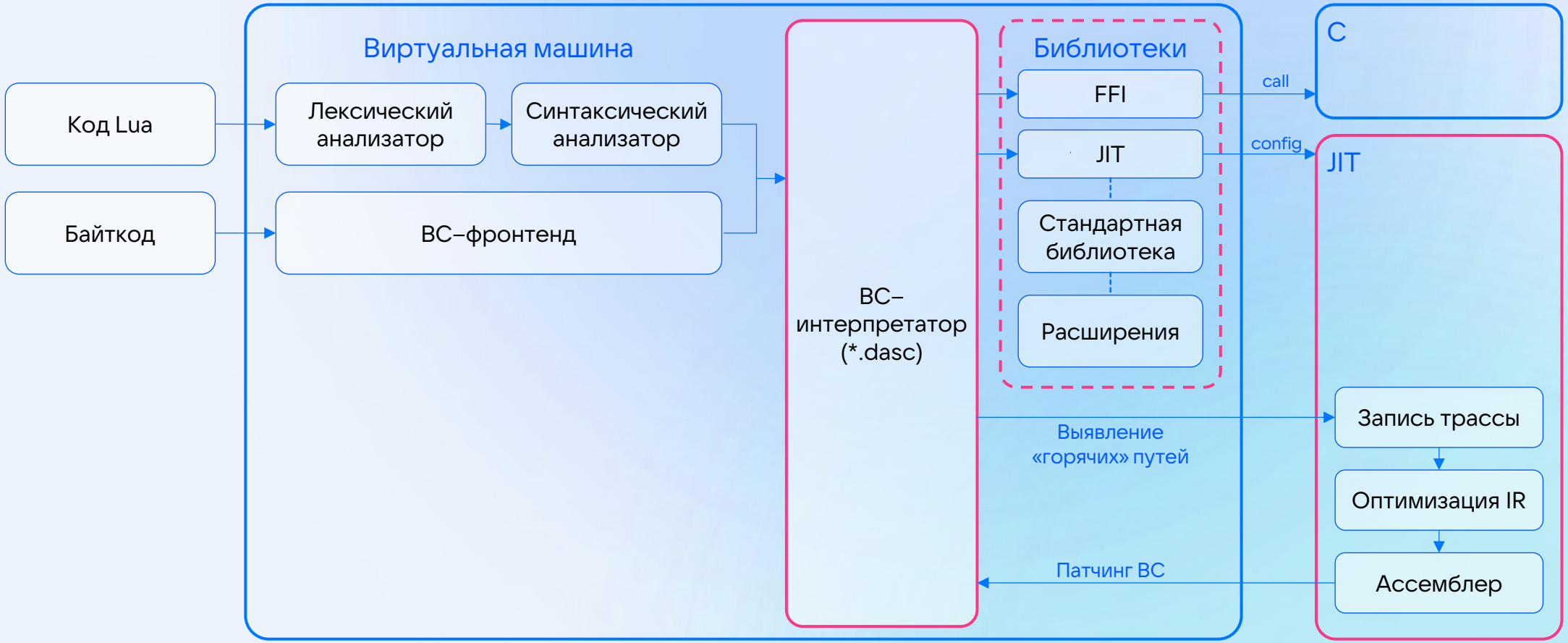
tech

LuaJIT: Внутреннее устройство



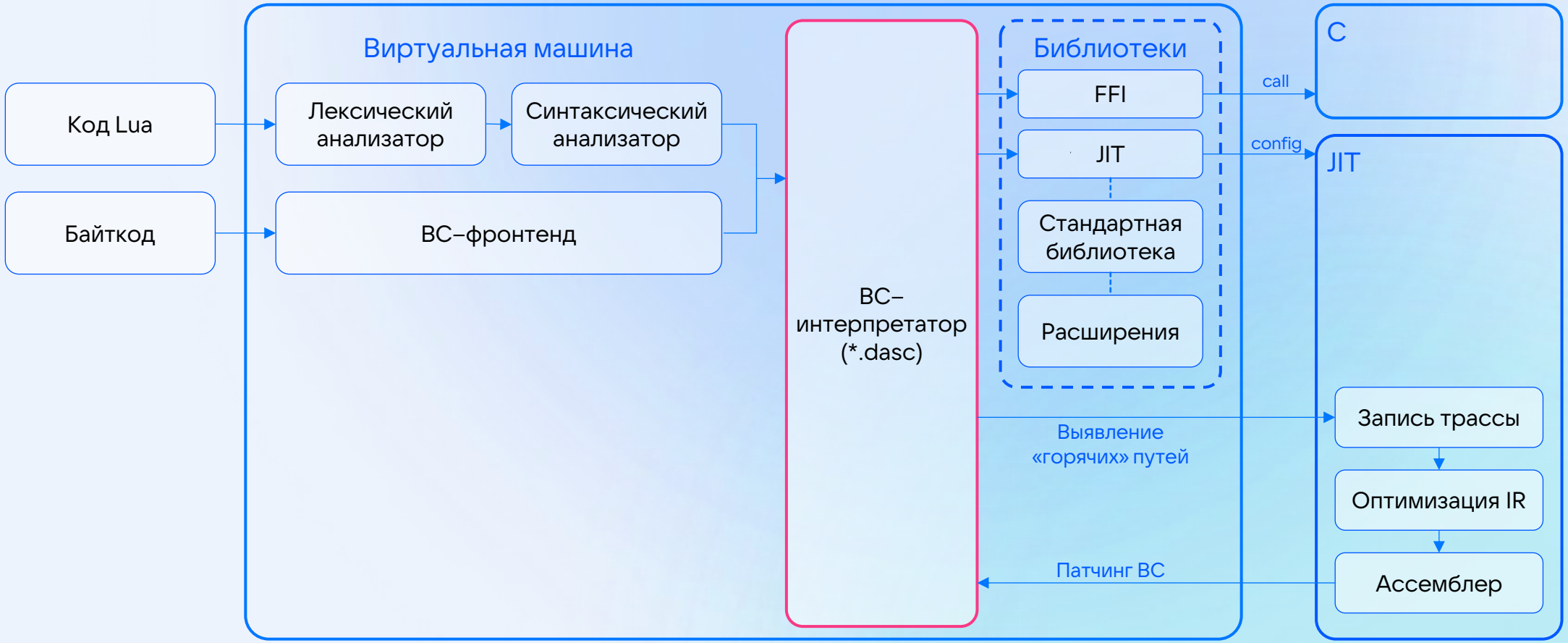
tech

LuaJIT: Внутреннее устройство



tech

LuaJIT: Интерпретатор



LuaJIT: Интерпретатор

Как выглядит интерпретатор?

```
switch {
case BC_<NAME1>:
        …
case BC_<NAME2>:
        …
…
}
```

tech

LuaJIT: Интерпретатор

DynASM

Макро-ассемблер
↳который генерирует код
↳который генерирует код

```
case BC_CAT:
    |  ins_ABC
    |  mov L:CARG1, SAVE_L
    |  mov L:CARG1->base, BASE
    |  lea CARG2, [BASE+RC*8]
    |  mov CARG3d, RCd
    |  sub CARG3d, RBd
```

vk tech

LuaJIT: Интерпретатор

DynASM

Макро-ассемблер
↳который генерирует код
↳который генерирует код

```
case BC_CAT:
    |  ins_ABC
    |  mov L:CARG1, SAVE_L
    |  mov L:CARG1->base, BASE
    |  lea CARG2, [BASE+RC*8]
    |  mov CARG3d, RCd
    |  sub CARG3d, RBd
```

vk tech

LuaJIT: Интерпретатор

DynASM

Макро-ассемблер
↳который генерирует код
↳который генерирует код

```
case BC_CAT:
  |  ins_ABC
  |  mov L:CARG1, SAVE_L
  |  mov L:CARG1->base, BASE
  |  lea CARG2, [BASE+RC*8]
  |  mov CARG3d, RCd
  |  sub CARG3d, RBd
```

vk tech

LuaJIT: Интерпретатор

DynASM

Макро-ассемблер
↳который генерирует код
↳который генерирует код

```
case BC_CAT:
    |  ins_ABC
    |  mov L:CARG1, SAVE_L
    |  mov L:CARG1->base, BASE
    |  lea CARG2, [BASE+RC*8]
    |  mov CARG3d, RCd
    |  sub CARG3d, RBd
```

LuaJIT: Интерпретатор

DynASM

Макро-ассемблер
↳который генерирует код
↳который генерирует код

```
case BC_CAT:
    |  ins_ABC
    |  mov L:CARG1, SAVE_L
    |  mov L:CARG1->base, BASE
    |  lea CARG2, [BASE+RC*8]
    |  mov CARG3d, RCd
    |  sub CARG3d, RBd
```

vk tech

LuaJIT: Интерпретатор

DynASM

```
Макро-ассемблер
 ↳который генерирует код
  ↳который генерирует код
```

```
…
case BC_CAT:
    dasm_put(
        Dst,
        3783,
        Dt1(->base),
        Dt1(->base),
        GG_G2DISP
    );
…
```

tech

LuaJIT: Интерпретатор

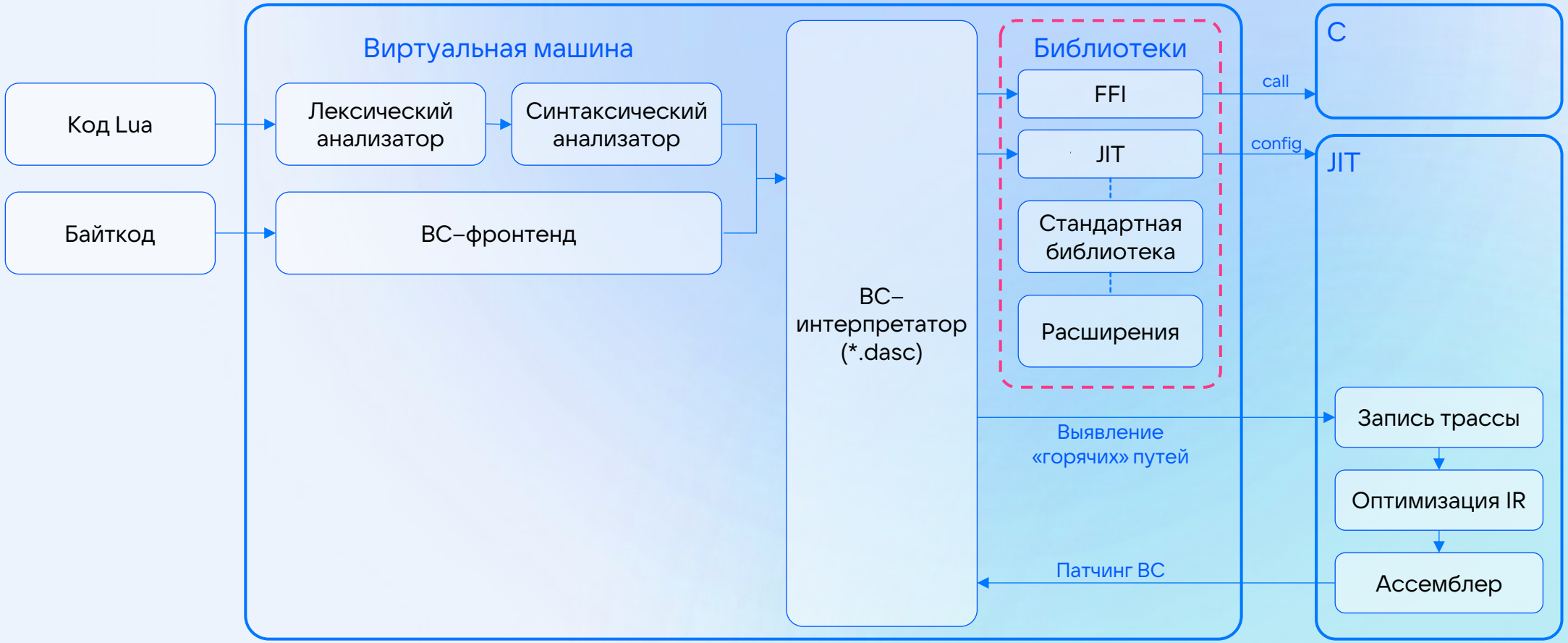
DynASM

```
Макро-ассемблер
 ↳который генерирует код
  ↳который генерирует код
```

```
_lj_BC_CAT:
        .long 0xd3587e11,0x92401f9c
.long 0xf90012f3,0xcb110382
        .long 0x8b1c0e61,0xaa1703e0
.long 0xf90007f5
        bl _lj_meta_cat
        .long 0x385ff2b1,0xf94012f3
,long 0xb500cbe0,0xf8717a68
        .long 0xf83b7a68,0xb84046b0
.long 0x8b300ec9,0xd3483e1b
        .long 0xf947f528,0xd3507e1c
.long 0xd61f0100
```

tech

LuaJIT: Стандартная библиотека



tech

LuaJIT: Стандартная библиотека

- Встроенные (builtin) функции: loadstring(), pcall(), getmetatable(), etc...
- Стандартная библиотека: os, table, math, string, etc...
- Библиотека LuaJIT FFI
- Библиотека для управления JIT-компилятором

LuaJIT: «Fast» functions

Интерпретатор

```
|.ffunc tonumber
|  // Only handles the number case inline
|  cmp NARGS:RDd, 1+1;  jne ->fff_fallback
|  mov RB, [BASE]
|  checknumber RB, ->fff_fallback
|  mov PC, [BASE-8]
|  mov [BASE-16], RB
|  jmp ->fff_res1
```

Fallback

```
LJLIB_ASM(tonumber)      LJLIB_REC(.)
{
...
if (tviscdata(o)) {
                CTState *cts = ctype_cts(L);
 ... 54 more SLOC ...
}
```

LuaJIT: «Fast» functions

Интерпретатор

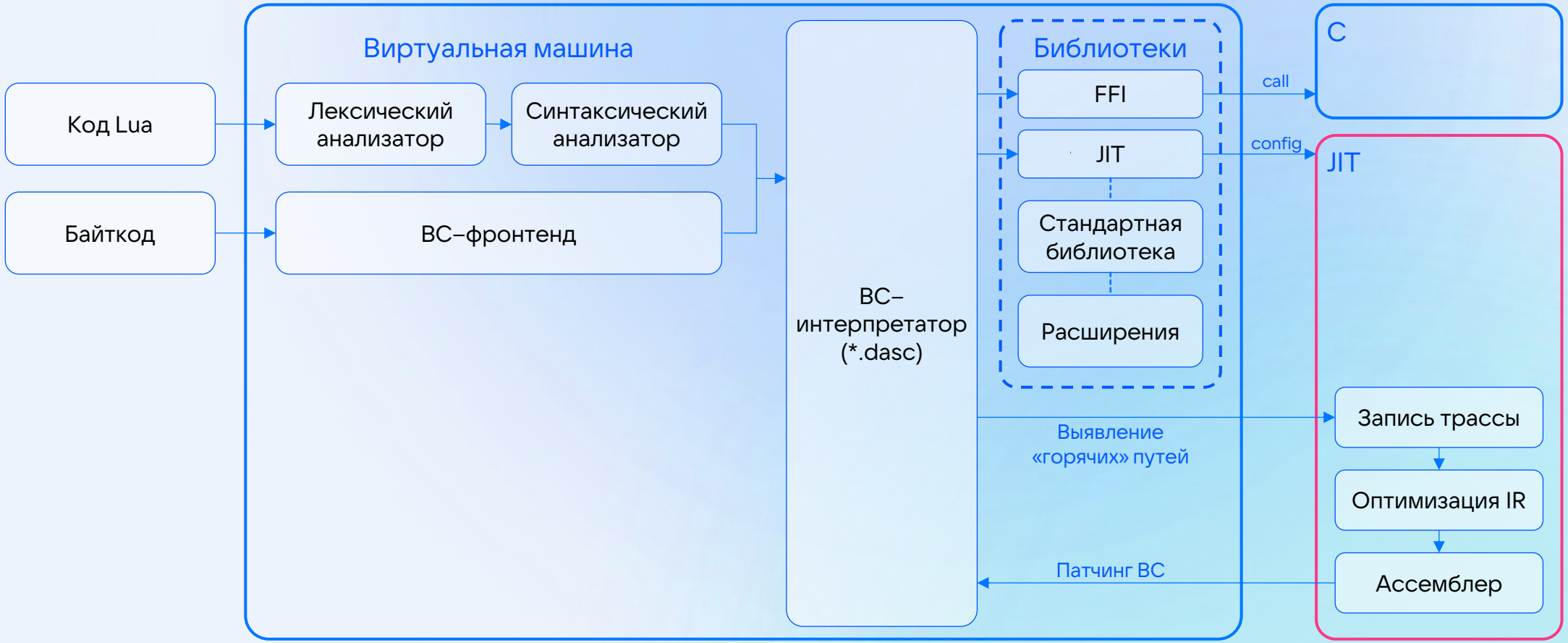
```
|.ffunc tonumber
|  // Only handles the number case inline
|  cmp NARGS:RDd, 1+1;  jne ->fff_fallback
|  mov RB, [BASE]
|  checknumber RB, ->fff_fallback
|  mov PC, [BASE-8]
|  mov [BASE-16], RB
|  jmp ->fff_res1
```

Fallback

```
LJLIB_ASM(tonumber)      LJLIB_REC(.)
{
...
if (tviscdata(o)) {
                CTState *cts = ctype_cts(L);
 ... 54 more SLOC ...
}
```

vk tech

LuaJIT: Компилятор



LuaJIT: Как работает JIT

ВМ

JIT

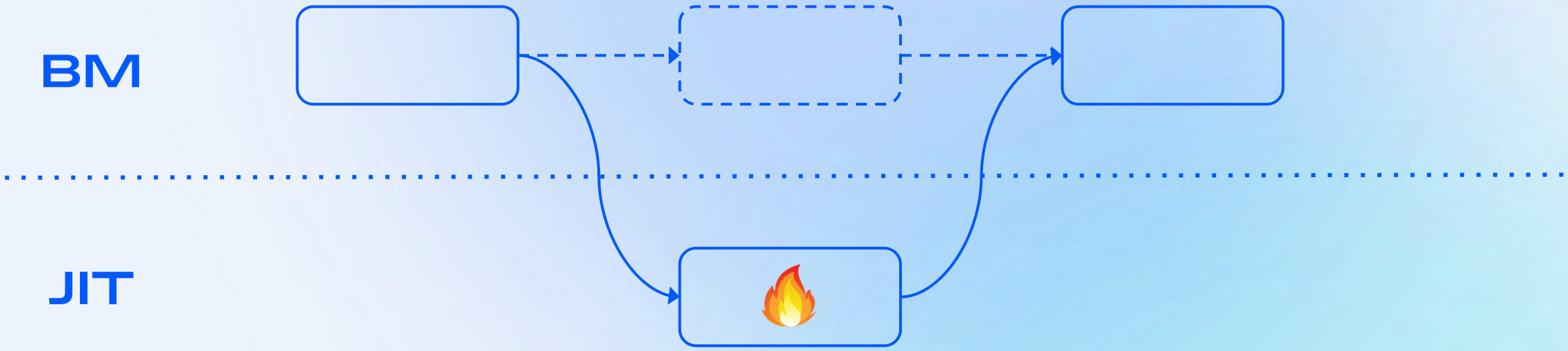
LuaJIT: Как работает JIT

LuaJIT: Как работает JIT

ВМ

JIT

LuaJIT: Как работает JIT



LuaJIT: Поиск «горячих» инструкций

Специальные инструкции в байткоде

FORL, LOOP, ITERL, FUNCF, FUNCV.

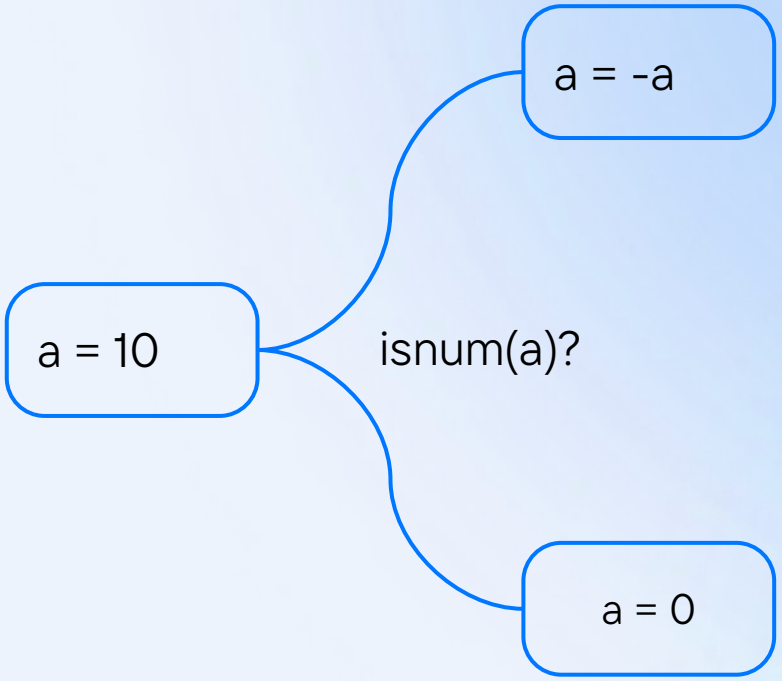
Каждое исполнение любой из этих инструкций модифицирует хеш-таблицу счетчиков.

LuaJIT: Трассировка

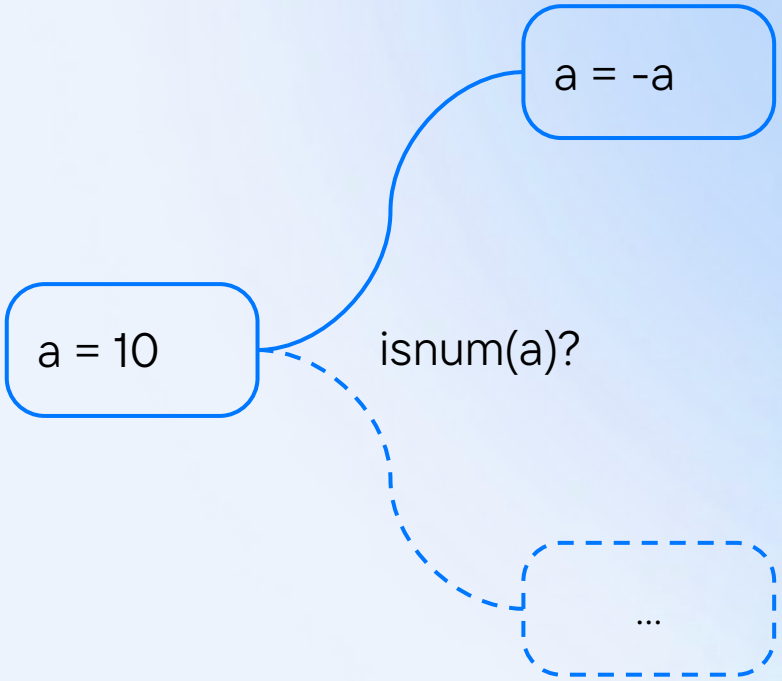
a = 10

tech

LuaJIT: Трассировка

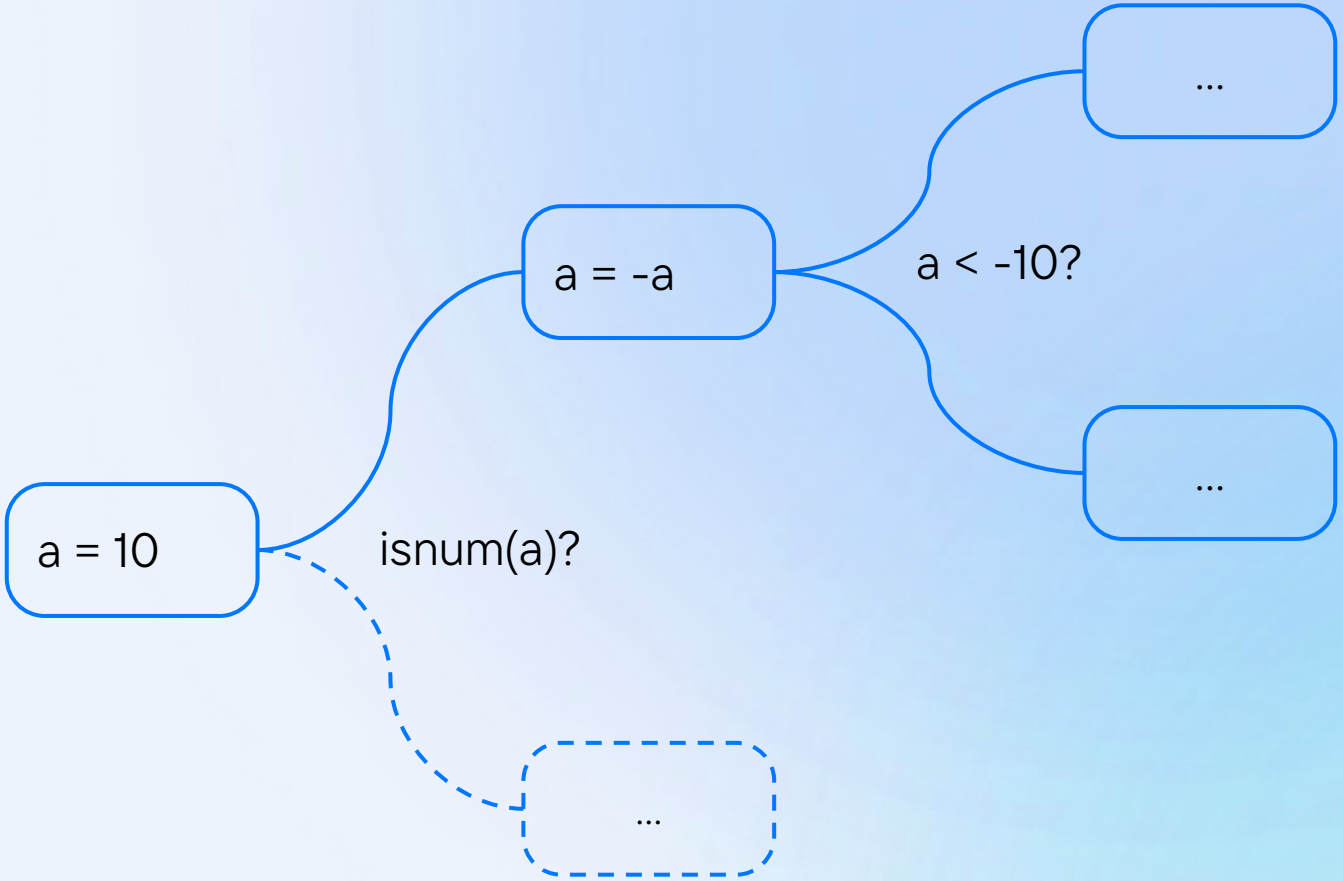


LuaJIT: Трассировка



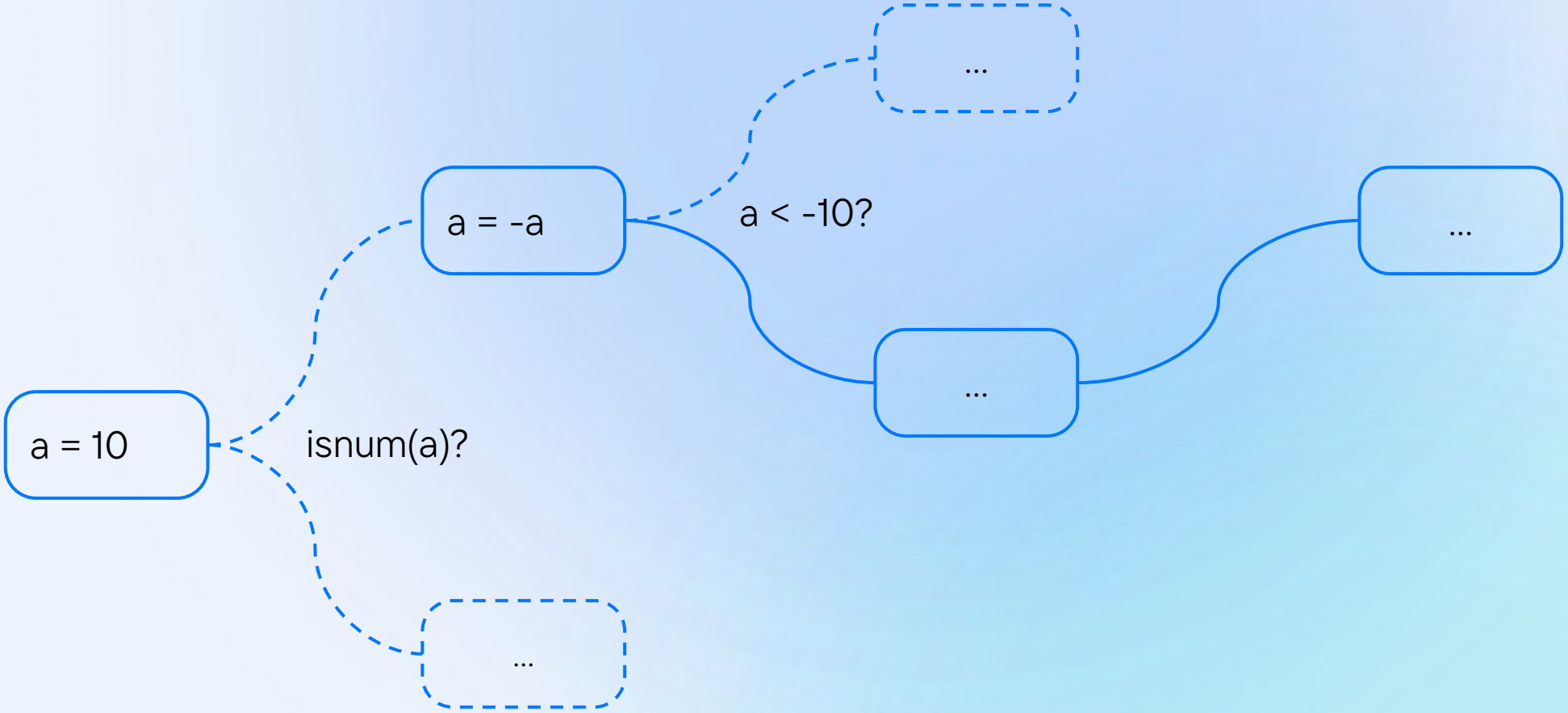
tech

LuaJIT: Трассировка



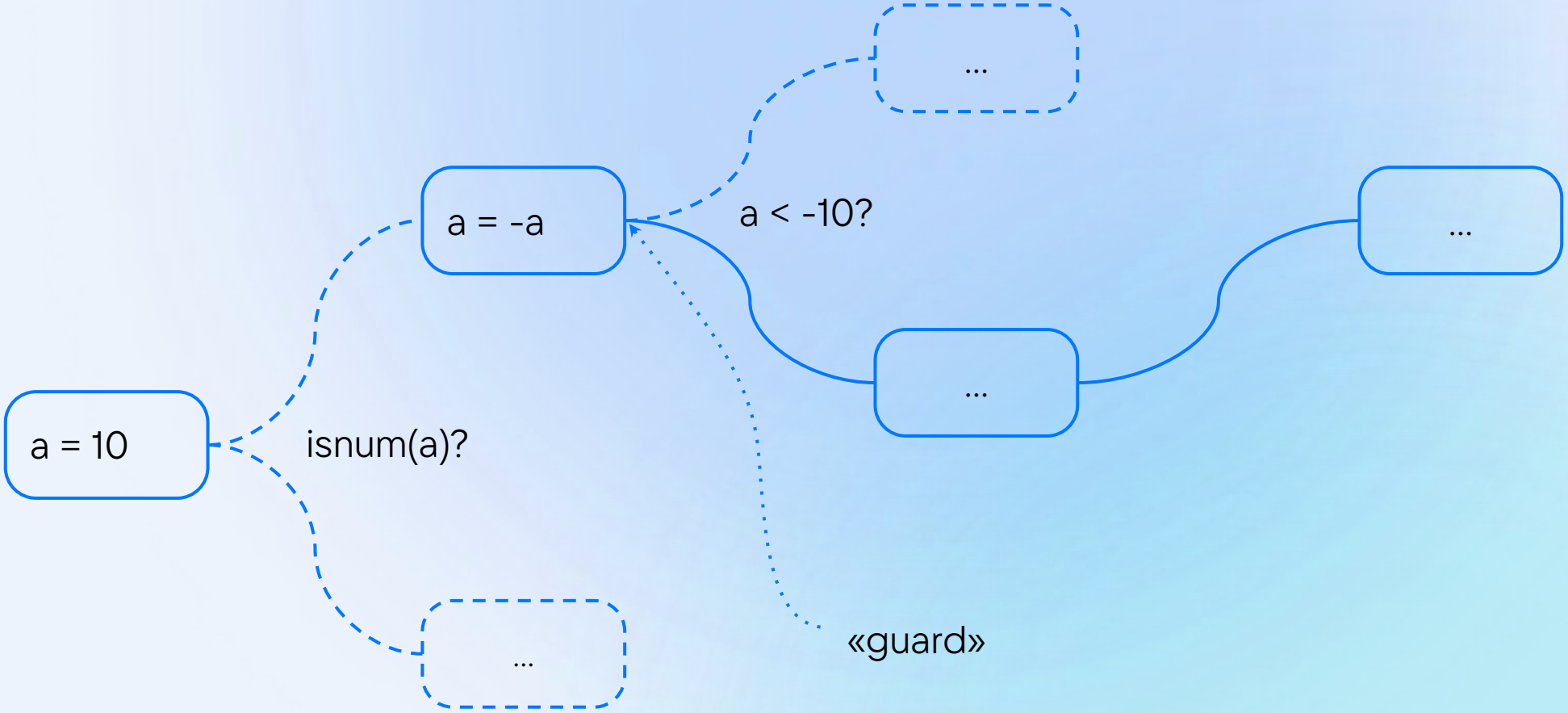
tech

LuaJIT: Трассировка



tech

LuaJIT: Трассировка



tech

LuaJIT: Снапшоты

Позволяют переходить между ВМ и трассами

Дают отображение между регистрами и слотами стека ВМ

Синхронизируются только изменившиеся значения

LuaJIT: Зачем нам нужен IR?

```
local sum = 0
for i = 1, 4 do
    sum = sum + i
end
```

LuaJIT: Зачем нам нужен IR?

```
local sum = 0
for i = 1, 4 do
    sum = sum + i
end
```

```
---- TRACE 1 start
0011  ADDVV    0   0   4
0012  FORL     1 => 0011
```

tech

LuaJIT: Зачем нам нужен IR?

```
local sum = 0
for i = 1, 4 do
    sum = sum + i
end
```

```
---- TRACE 1 start
0011  ADDVV    0   0   4
0012  FORL     1 => 0011
```

```
---- TRACE 1 IR
0001     int SLOAD  #3    I
0002     u8  XLOAD  [0x101004521]  V
0003     int BAND   0002  +12
0004 >   int EQ     0003  +0
0005 >   int SLOAD  #2    T
0006 >+  int ADDOV  0005  0001
0007  +  int ADD    0001  +1
0008 >   int LE     0007  +4
0009 ------ LOOP ------------
0010     u8  XLOAD  [0x101004521]  V
0011     int BAND   0010  +12
0012 >   int EQ     0011  +0
0013 >+  int ADDOV  0007  0006
0014  +  int ADD    0007  +1
0015 >   int LE     0014  +4
0016     int PHI    0007  0014
0017     int PHI    0006  0013
```

LuaJIT: Оптимизации в IR

- General
  - Control-Flow Specialization
  - Hash Key Specialization
  - Fast Function Inlining
  - Bytecode Patching
- Fold Engine
- Common Subexpression Elimination
- Array Bounds Check Elimination
- Narrowing
- Memory Access Optimizations
- Loop Optimizations
- FP Value Splitting and Soft-Float Calls
- Allocation Sinking and Store Sinking

LuaJIT: Пример работы

Lua

```
fun.range(5):
    map(function(x)
        return x^2
    end):
    reduce(fun.operator.add, 0)
```

tech

LuaJIT: Пример работы

Lua

```
fun.range(5):
    map(function(x)
        return x^2
    end):
    reduce(fun.operator.add, 0)
```

Ассемблер

Один машинный цикл с 10 ассемблерными инструкциями:

```
-- skip some initialization code --
->LOOP:
0bcaffd0  movaps xmm5, xmm7
0bcaffd3  movaps xmm7, xmm1
0bcaffd6  addsd xmm7, xmm5
0bcaffda  ucomisd xmm7, xmm0
0bcaffde  jnb 0x0bca0024        ->5
0bcaffe4  movaps xmm5, xmm7
0bcaffe7  mulsd xmm5, xmm5
0bcaffeb  addsd xmm6, xmm5
0bcaffef  jmp 0x0bcaffd0        ->LOOP
---- TRACE 1 stop -> loop
```

LuaJIT: Нарушение DRY

- «Fast» functions
- Обход таблиц
- Индексирование таблиц
- Листер байт-кода
- Функция jit.dump() дублирует часть компилятора

Как и зачем LuaJIT нарушает DRY?
Антон Солдатов

LuaJIT: Если хочется ещё

What I learned from LuaJIT
Слава Егоров

https://mrale.ph/talks/vmss16/

LuaJIT: обо всем и ни о чем
Максим Кокряшкин

https://www.youtube.com/watch?v=pDTwyL49JYY

Lua – идеальный встраиваемый язык
Антон Солдатов

Podlodka Podcast #392

Известные проблемы LuaJIT

Вспомогательный код для сборки проекта LuaJIT содержит проблемы работы с памятью, эти проблемы не будут исправлены

01

Грязные чтения при работе со строками

02

Хрупкие парсеры байткода и C-объявлений для FFI

03

Есть известные неисправленные проблемы с неопределенным поведением с точки зрения языка C

04

Необдуманное использование функций библиотеки *debug* может приводить к крешам неожиданным эффектам

05

Известные проблемы

Вспомогательный код для сборки проекта LuaJIT содержит проблемы работы с памятью, эти проблемы не будут исправлены

01

Грязные чтения при работе со строками

02

Хрупкие парсеры байткода и C-объявлений для FFI

03

В LuaJIT есть известные неисправленные проблемы с неопределенным поведением с точки зрения языка C

04

В LuaJIT и PUC Rio Lua необдуманное использование функций библиотеки *debug* может приводить к крешам неожиданным эффектам

05

LuaJIT: Вызовы для тестирования

01 Реализация с помощью низкоуровневых языков с ручным управлением памятью

02 Платформозависимый код

03 Нарушение принципа Don’t Repeat Yourself

04 Два режима работы: интерпретатор и компилятор

05 Восстановление с трассы в интерпретатор (снапшоты)

06 Оптимизации в интерпретаторе и компиляторе

07 Известные неисправленные проблемы

08 Два API: C и Lua

03
LuaJIT в Tarantool

Проблемы LuaJIT в Tarantool

- Крошился на продакшене, баги невоспроизводимы
- Страх добавления новых изменений
- Бэкпорт патчей:
  - Патчи в ванильной версии без репродьюсера.
  - Некорректные патчи в ванильной версии.
- Сложные отношения с ментейнером LuaJIT

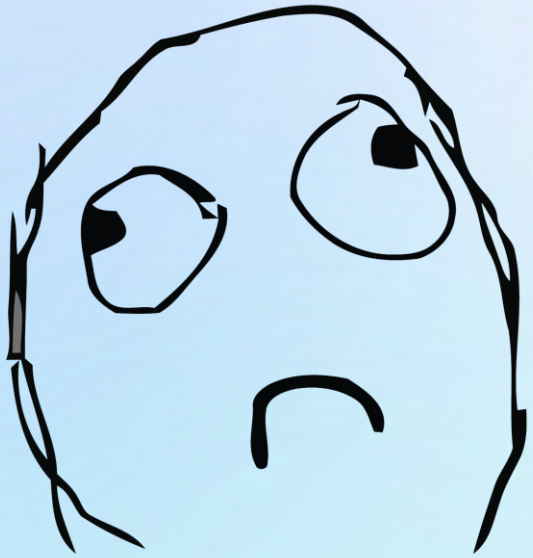
```
> The response from the Tarantool folk is shown below.

*Sigh* I was really hoping I would never have to hear that name
again.

--Mike
```

tech

Работаем над LuaJIT в Tarantool

- Отсутствие тестов в апстриме компенсируем регрессионным набором тестов:
  - Регрессионные тесты проекта PUC Rio Lua
  - lua-Harness от Франсуа Перра (François Perrad)
  - Собственные тесты для LuaJIT
- Покрытие кода тестами: 78% для строк, 67% для ветвлений
- Аккуратно бэкпортим патчи
- Работа с багфиксами в режиме «upstream first»
- Каждая новая строка в наших патчах покрывается тестами
- Запуск регрессионных тестов на каждый PR

Но тестирования всё ещё недостаточно!



04
Фаззинг LuaJIT

Дизайн тестов

Тестирование с точки зрения API

- Функции C API
- Функции Lua API

01

Тестирование с точки зрения входных данных

- Байткод
- Код Lua

02

Lua API и C API

Lua API	Стек Lua	Индекс		C API
				lua_setglobal(L, “foo”)
foo = 100 + 200	300	1	-1	lua_arith(L, LUA_OPADD)
	200	2	-2	lua_pushinteger(L, 200)
	100	1	-1	lua_pushinteger(L, 100)

tech

C API: Индикатор функции

Сколько элементов функция «забирает» со стека?

Бросает ли функция исключение?

[-o, +p, x]

Сколько элементов функция добавляет на стек?

lua_pushnumber

```
void lua_pushnumber (lua_State *L, lua_Number n);
```

[-0, +1, -]

Pushes a number with value n onto the stack.

Lua Reference Manual: 3.7 – Functions and Types

Тесты для функций C API

Генератор случайных данных на стеке

Выполнение случайных Lua C функций

Тестовый оракул – индикатор функции

PUC Rio Lua: покрытие 91% кода можно добиться с помощью фаззинга только Lua C API

Тесты для функций Lua API

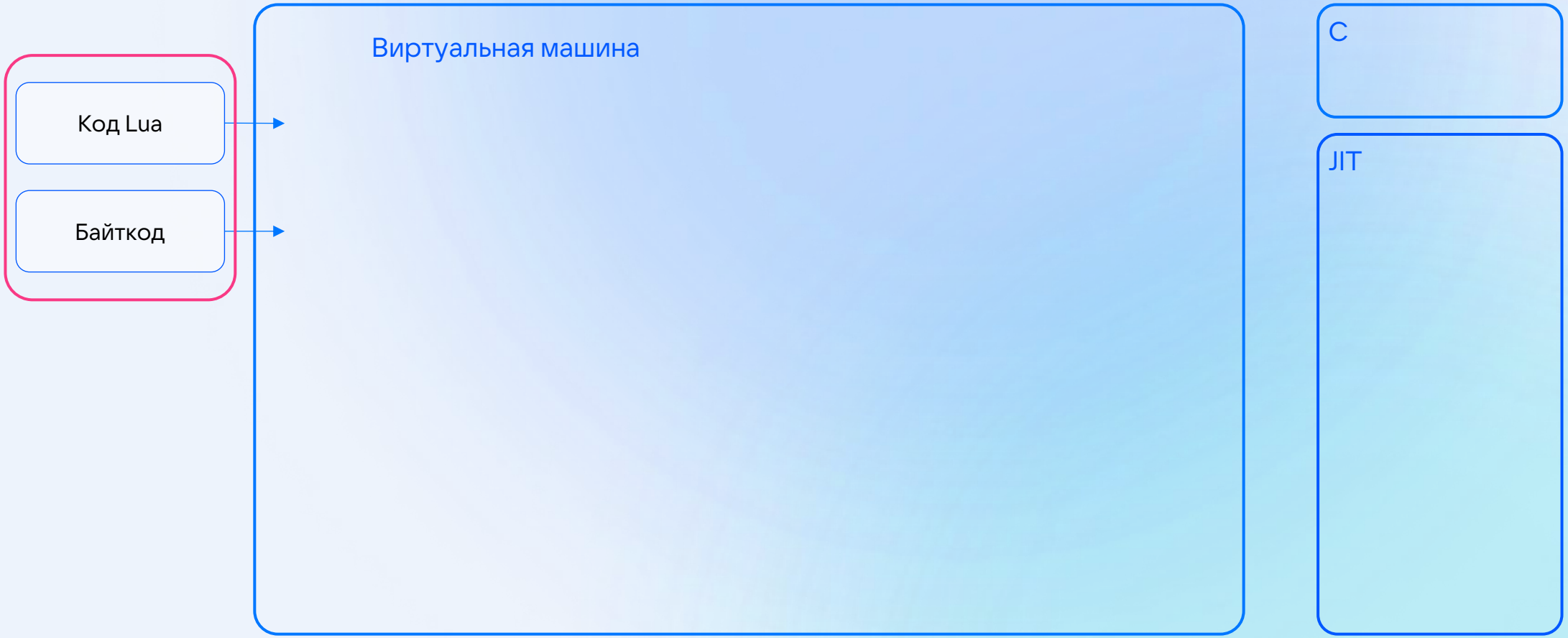
- Тесты для каждой функции стандартной библиотеки
- Анализ исправленных проблем
- Проблема тестового оракула
- Нативный фаззинг для Lua – luzer

Фаззер для Lua API - luzer

- Фаззинг Lua кода и нативных расширений на C
- Под капотом libFuzzer
- Инструментация:
  - Статическая для C - SanitizerCoverage
  - Динамическая для Lua - хук на строки
- Включает FDP (Fuzzing Data Provider)
- Включен в состав Crusher от ИСП РАН
- В планах добавить поддержку в OSS Fuzz

Делаем фаззер для Lua
на основе LibFuzzer, Heisenbug

LuaJIT: Входные данные



tech

Генератор Lua программ

Грамматика языка Lua описана в формате EBNF

Транслировали EBNF в схему в формате Google Protocol Buffers

Программа генерируется с помощью схемы случайным образом и protobuf сериализуется в код на Lua с помощью LibProtoBufMutator

Улучшаем генератор Lua кода

Контроль полноты
с помощью анализа покрытия
кода

01

Контроль работы LuaJIT
с помощью метрик

- ошибки синтаксического анализатора
- количество записанных трасс
- причины выхода с трасс

02

Проблемы сериализации

Синтаксические ошибки:

- некорректное использование return, break (42% ошибок!)
- некорректное использование vararg: cannot use '...' outside a vararg function
- использование некорректного индекса: table index is {nil, NaN}
- некорректные значения для оператора for
- использование ключевых слов
- ...

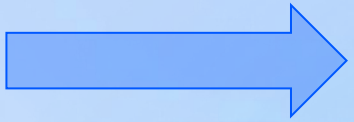
Семантические ошибки:

- «attempt to index» (16%)
- «attempt to perform arithmetic on» (12%)
- «attempt to call» (11%)
- «attempt to concatenate» (3%)

Таймауты из-за зацикливаний и рекурсий в коде

Исправление таймаутов

```
while (true) do
    foo = 'bar'
    ...
end
```



```
counter_0 = 0
while (true) do
    if counter_0 > 5 then
        break
    end
    ...
end
```

Program Reconditioning: Avoiding Undefined Behaviour When Finding and Reducing Compiler Bugs

BASTIEN LECOEUR, Imperial College London, UK
HASAN MOHSIN, Imperial College London, UK
ALASTAIR F. DONALDSON, Imperial College London, UK

Исправление семантики

true + false -- attempt to perform arithmetic on a boolean value

Исправление семантики

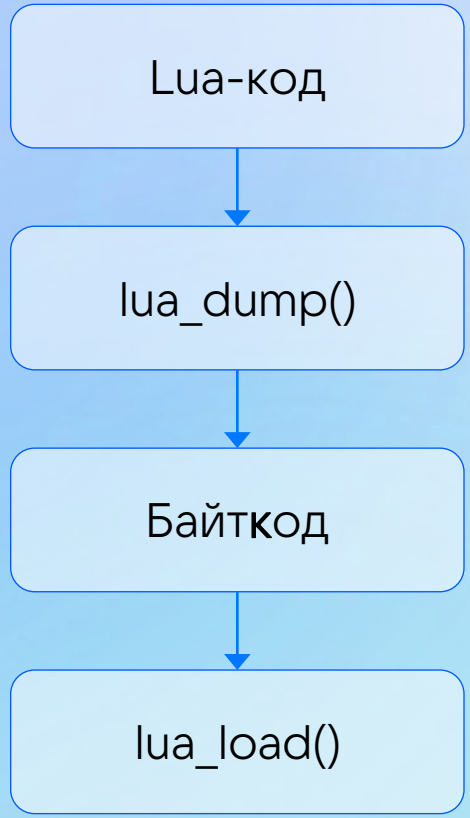
true + false -- attempt to perform arithmetic on a boolean value

```
always_number = function(v)
        return tonumber(val) or 10
end
debug.setmetatable(true, { __add = function(v1, v2)
        return always_number(v1) + always_number(v2)
end })
true + false -- 20
```

VK tech

Тестирование ВС-фронтенда

- Сделать генератор байткода сложно!
- Трансляция Lua кода в байткод: lua_dump() / string.dump()
- Решение: сгененируем байткод с помощью рантайма!



Интеграция в процесс разработки

Tarantool и PUC Rio Lua интегрированы в OSS Fuzz

Автоматизация рутины:

- оформление тикета с артефактами
- верификация исправленных багов в тикетнице
- минимизация тесткейса

Фаззинг LuaJIT использует ClusterFuzz

Планы

- Устранение недетерминированного поведения
- Инструментация кода, написанного вручную на ассемблере
- Фаззинг LuaJIT FFI, регистрового аллокатора, сборщика мусора
- Использование верификации с помощью моделей?

Результаты

Снижение количества инцидентов в продакшене Tarantool

Положительное влияние на сертификацию Tarantool во ФСТЭК

Автономное тестирование

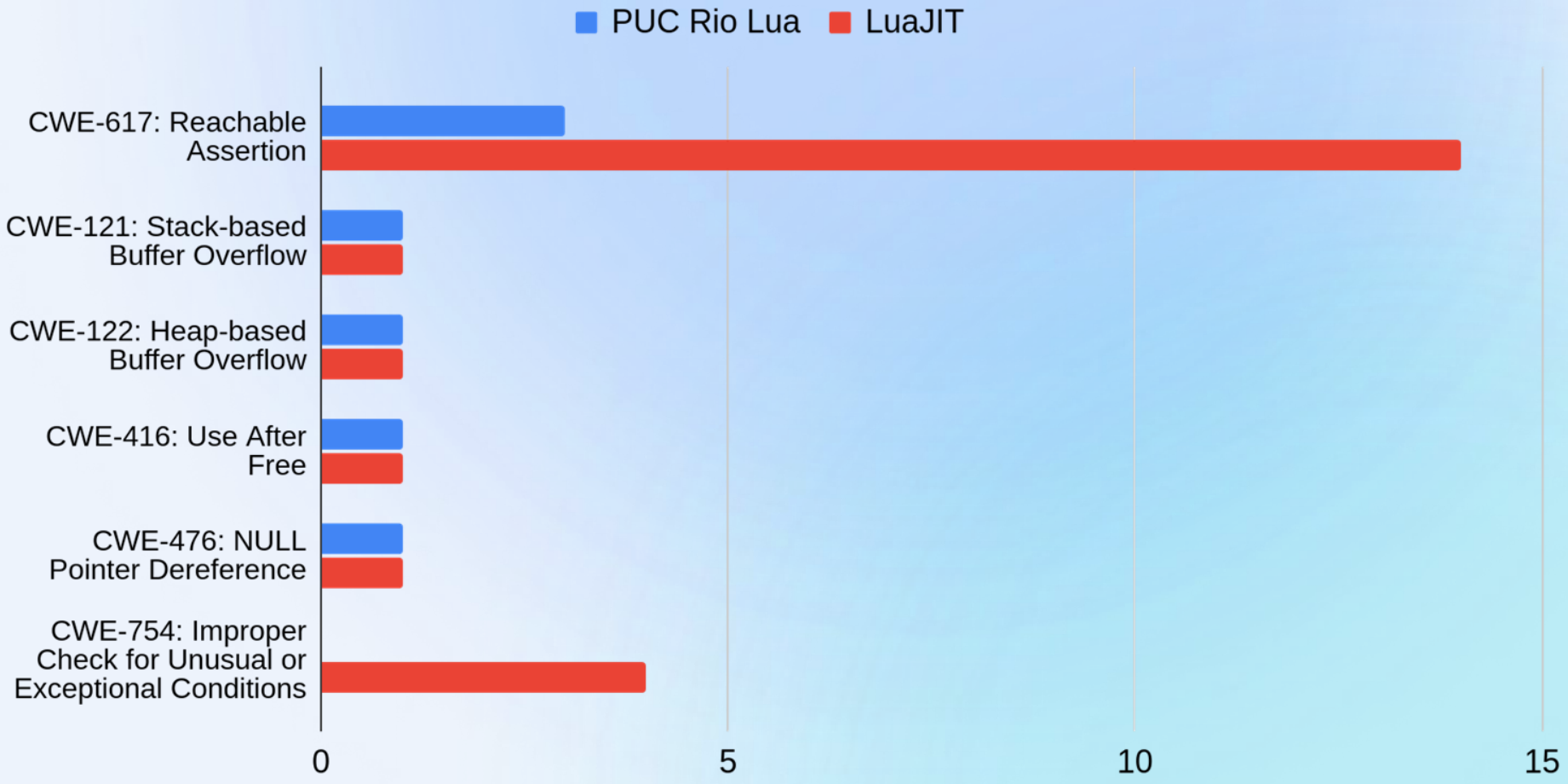
Переиспользование тестов для разных реализаций за счёт единого API:

- Повышение надежности и корректности кода Tarantool / LuaJIT / PUC Rio Lua
- Shift left за счёт непрерывного фаззинга PUC Rio Lua, LuaJIT, Tarantool's LuaJIT
- Благодарности от ментейнеров Mike Pall и Roberto Ierusalimsky за репорты дефектов

Покрытие LuaJIT фаззинг-тестами: 61% по строкам

Покрытие PUC Rio Lua фаззинг-тестами: 99% по строкам

Трофеи фаззинга



vk tech

Результаты

Код Tarantool's LuaJIT

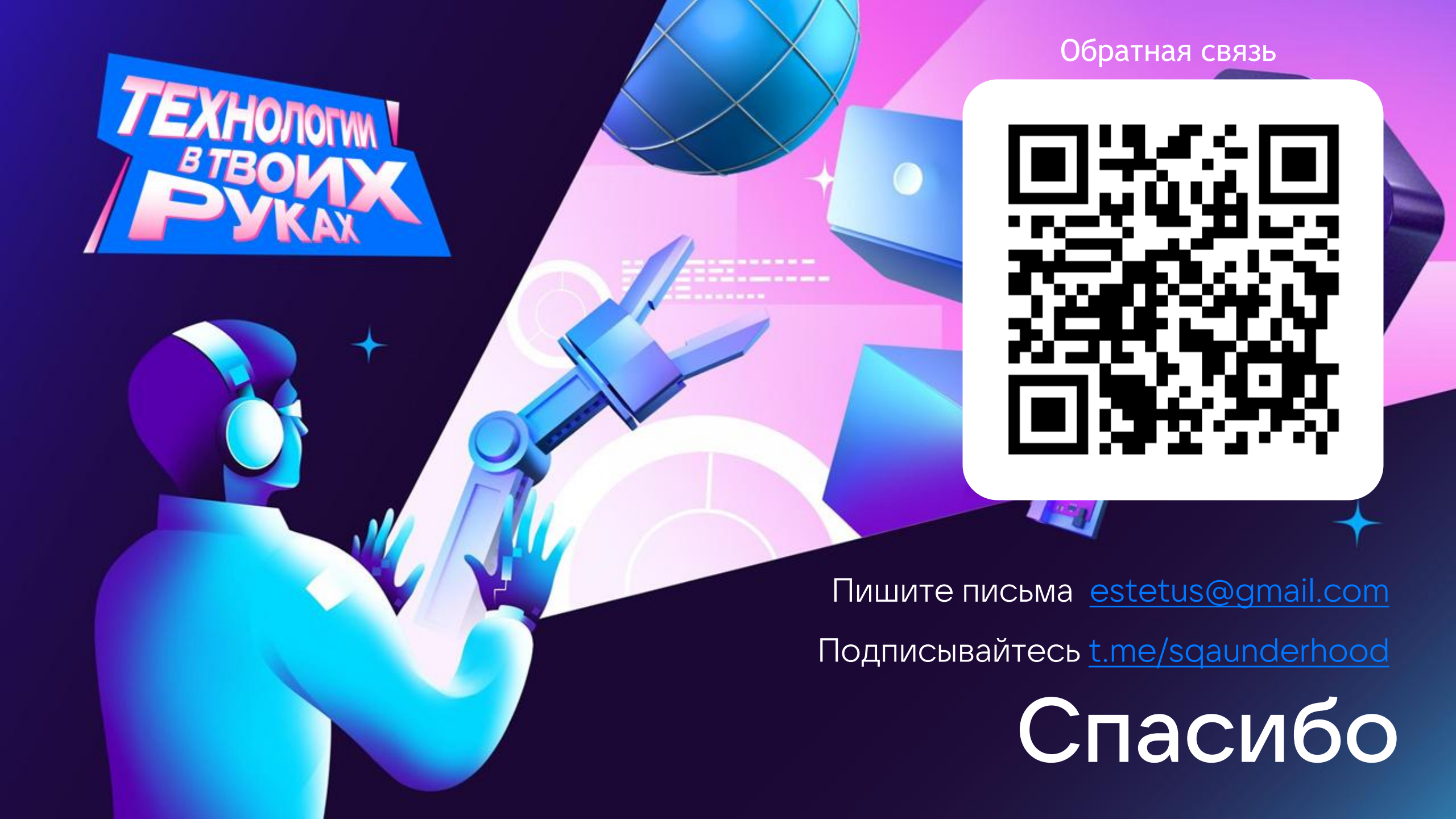
tarantool/luajit

Код тестов

ligurio/lua-c-api-tests

Код luzer

ligurio/luzer



Обратная связь

Пишите письма estetus@gmail.com

Подписывайтесь t.me/sqaunderhood

Спасибо