
Яндекс

Яндекс

Pandora: нагрузочные тесты в виде кода

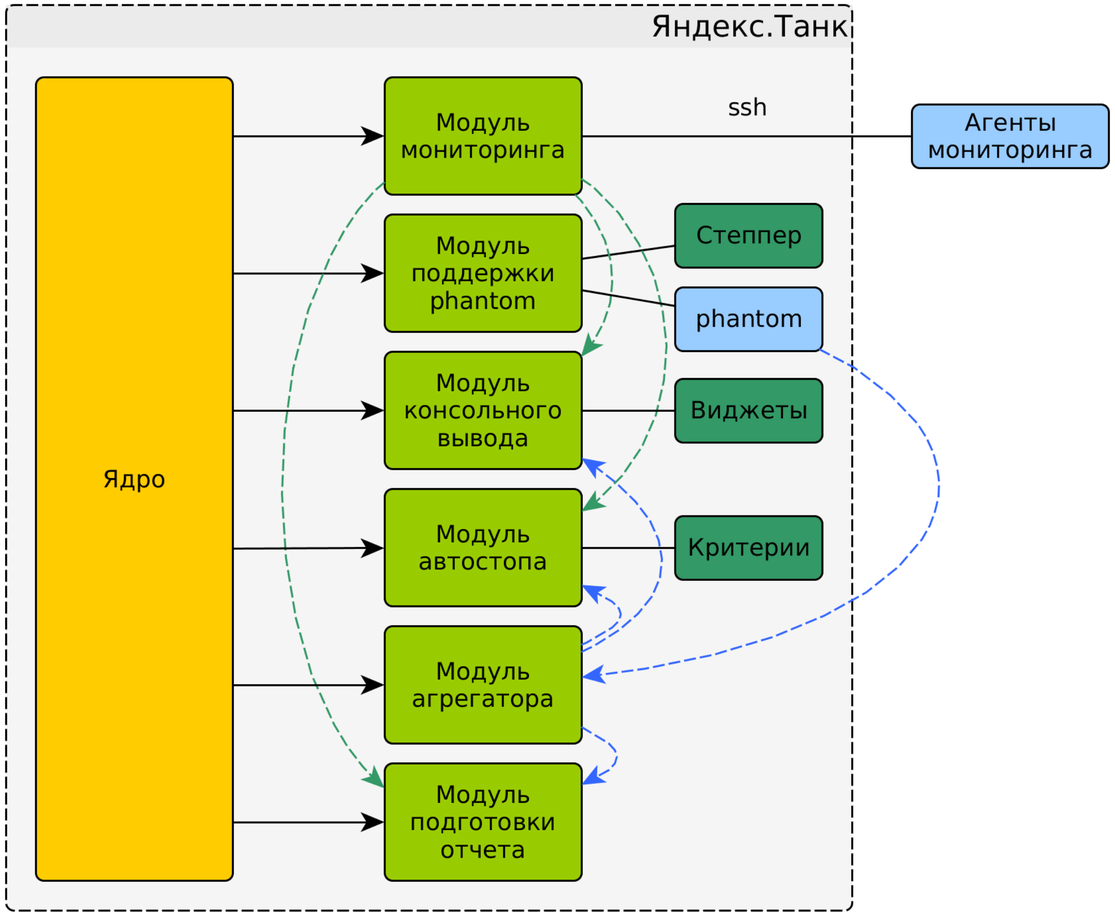
Алексей Лавренюк

Disclaimer: я не продавец пушек

> У меня нет цели продать вам Pandora. Мне лично она нравится (но я автор). Отнеситесь к ней с разумной осторожностью, попробуйте на своих кейсах. Решите сами, нужна она вам или нет.

Pandora — еще один генератор нагрузки. Зачем?

Танк: ядро + плагины



Пушки для Яндекс.Танка (до появления Pandora)

Пушки для Yandex.Tank (из коробки):

- **Phantom**: обстрелять производительный и простой сервис на HTTP
- **JMeter**: простые сценарии и не HTTP
- **BFG**: сложные сценарии и не HTTP

Phantom

Плюсы:

- очень быстрый (~100000 RPS)
- патроны готовятся из логов продакшна
- стабильный

Минусы:

- только stateless протоколы, в основном, HTTP/1.x
- патроны готовятся до стрельбы, могут занимать десятки гигабайт и генерятся очень долго
- только Linux

JMeter

Плюсы:

- достаточно быстрый (~30000 RPS)
- кроссплатформенный
- много плагинов
- низкий порог входа, настраивается из GUI

Минусы:

- негибкий, 1 поток = 1 пользователь
- сценарии в GUI сильно ограничены, приходится писать код
- код на Groovy (= медленнее, чем Java)
- JMX + кучка файлов, сложно деплоить сценарии
- сам JMeter неудобно таскать (это много файлов + JVM)

BFG

Плюсы:

- **сценарии на Python** (можно хранить в Git)
- кроссплатформенный
- много библиотек (весь Python мир)
- производительности хватает для многих кейсов (завалить Postgress, ~15000 RPS)

Минусы:

- низкий уровень параллелизма (100 виртуальных пользователей)
- сложно деплоить сценарии

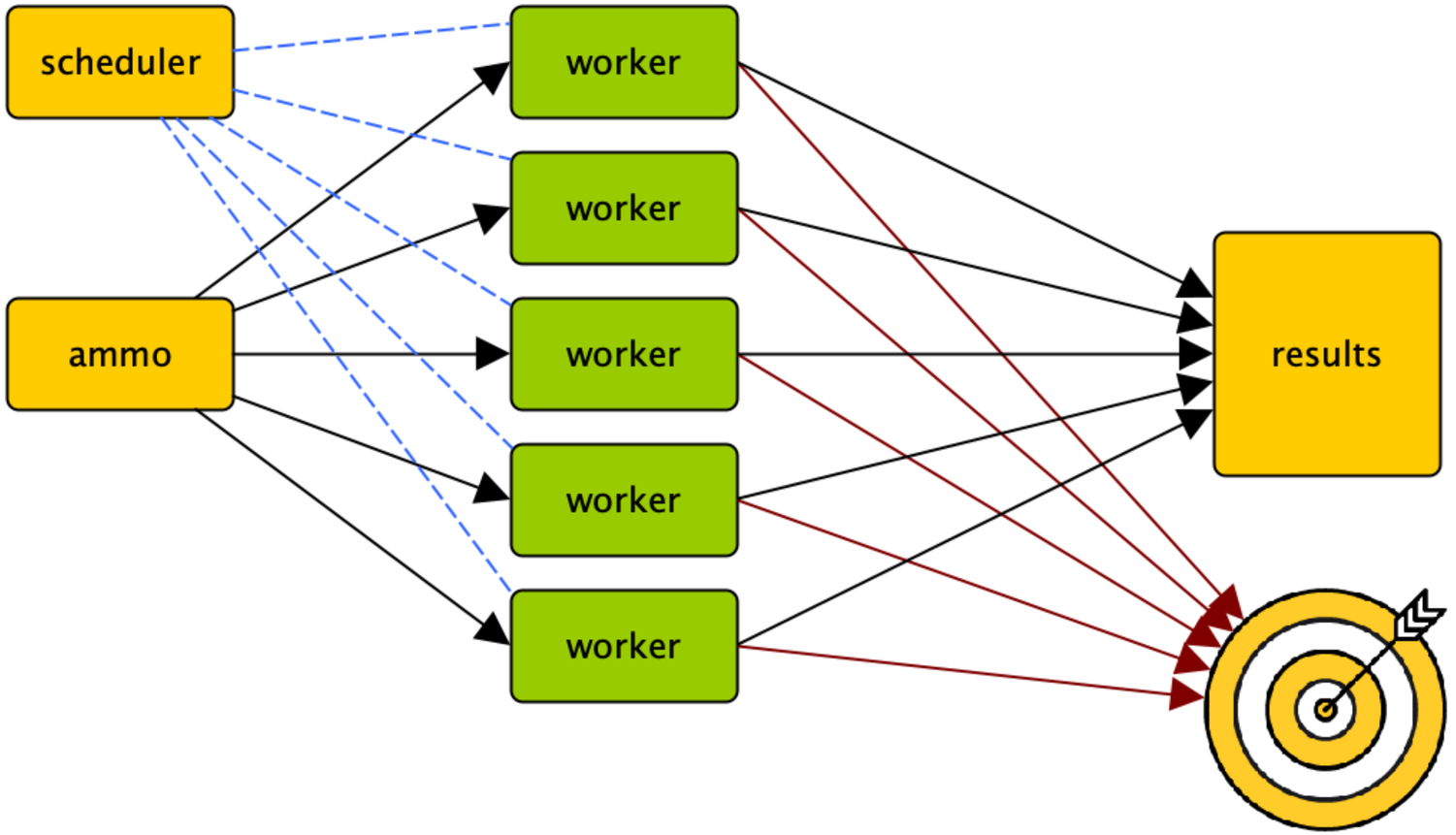
главная фича BFG: сценарии на Python

Чем удобны нагрузочные тесты в виде кода?

- сценарии нагрузочных тестов живут рядом с кодом приложения, любой разработчик может их посмотреть, поправить и добавить новые
- правка тестов в любимой IDE с подсветкой синтаксиса, линтером и подсказками
- тесты живут в системе контроля версий. Можно проводить code-review
- тестирование производительности встраивается в CI (сборка пушки, деплой, запуск тестов и анализ результатов)
- логирование в сценариях: удобно дебажить
- позволяет развиваться в сторону разработки

BFG — удобно.
когда его не хватает?

Архитектура пушки



Закон Литтла

$$L = \lambda W$$

L — число параллельных пользователей

λ — число запросов в секунду

W — среднее время ответа

2 sec * 5000 RPS = 10000 workers

Время ответа под нагрузкой растет, во время лагов растет ненадолго, но сильно. В пиках нужно больше пользователей.

Эксперименты по реализации воркеров BFG

Модели параллелизма в Python:

- **процессы**: тяжелые, накладные расходы на общение, много не создашь (~4x число ядер)
- **потоки**: GIL, упираемся в одно ядро
- **процессы + потоки**: этой конструкцией сложно управлять
- **процессы+asyncio**: медленно, нельзя гарантировать расписание, нельзя точно померить время, хаки в реализации, нужны специальные асинхронные библиотеки и асинхронные нагрузочные скрипты

в Python 100 пользователей — уже проблема

нужна новая пушка. Go?

Go: параллелизм на уровне языка

```
func shoot(ammo string) {
    // берем из ammo параметры и отправляем запрос
}

shoot("www.example.org/example.html")
```

Go: параллелизм на уровне языка

```
func shoot(ammo string) {
    // берем из ammo параметры и отправляем запрос
}

shoot("www.example.org/example.html")
// ждем завершения
shoot("www.example.org/example.html")
```

Go: параллелизм на уровне языка

```
func shoot(ammo string) {
    // берем из ammo параметры и отправляем запрос
}

go shoot("www.example.org/example.html")
// не ждем завершения
go shoot("www.example.org/example.html")
```

Go: параллелизм на уровне языка

```
func shoot(ammo string) {
    // берем из ammo параметры и отправляем запрос
}

// миллион запросов
for i := 0; i < 1000000; i++ {
    go shoot("www.example.org/example.html")
}
```

Go: messaging на уровне языка

```
func shoot(ammo string, result chan int) {
    // берем из ammo параметры и отправляем запрос
    result <- time // отправляем время в канал с результатами
}

for i := 0; i < 1000000; i++ {
    go shoot("www.example.org/example.html", result)
}
```

Go: messaging на уровне языка

```
func shoot(ammo string, result chan int) {
    // берем из ammo параметры и отправляем запрос
    result <- time // отправляем время в канал с результатами
}
result := make(chan int, 10000)
for i := 0; i < 1000000; i++ {
    go shoot("www.example.org/example.html", result)
}
```

Go: messaging на уровне языка

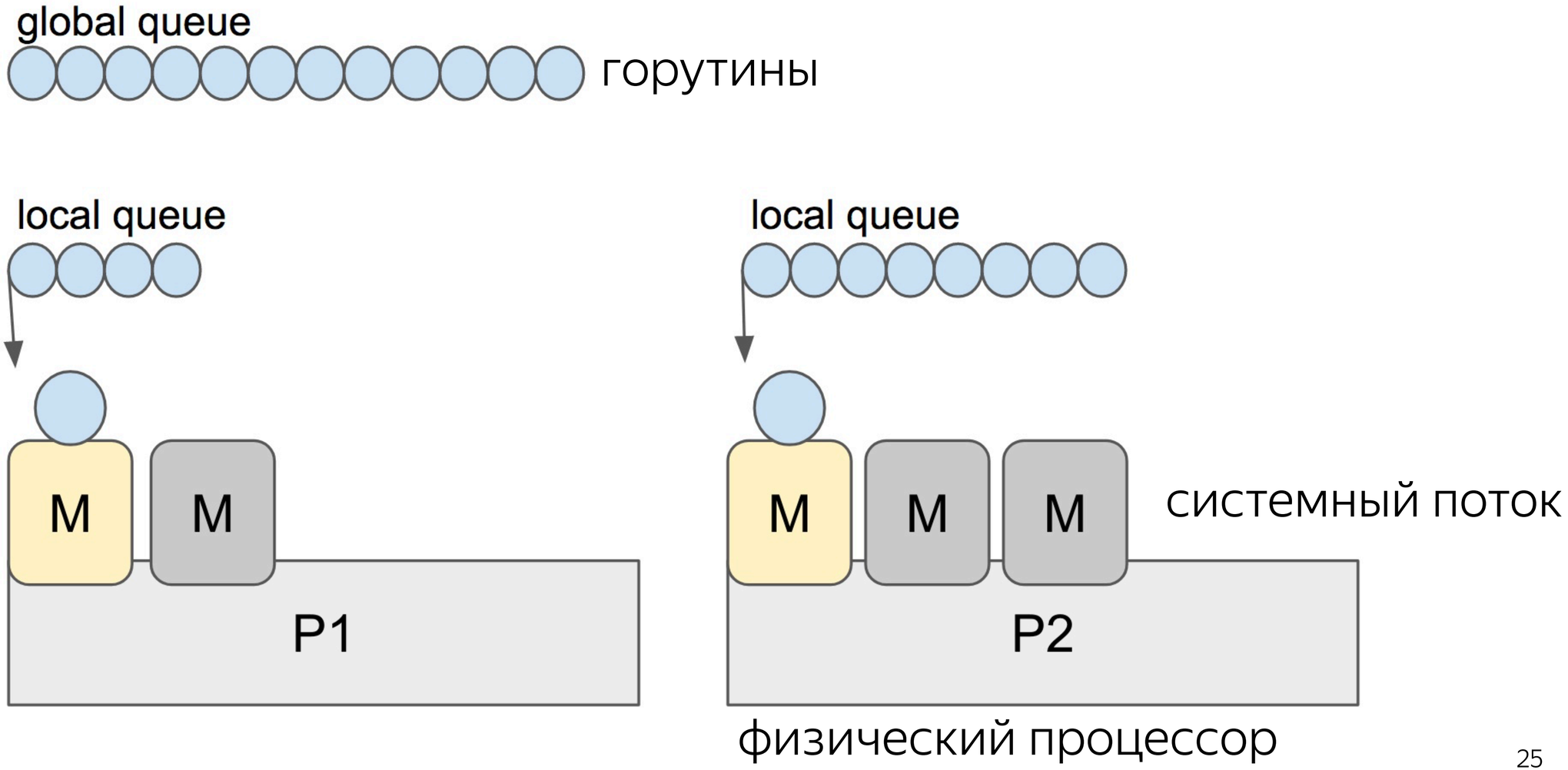
```
func shoot(ammo string, result chan int) {
    // берем из ammo параметры и отправляем запрос
    result <- time // отправляем время в канал с результатами
}
result := make(chan int, 10000)
for i := 0; i < 1000000; i++ {
    go shoot("www.example.org/example.html", result)
}

for r := range result {
    // обрабатываем результат r
}
```

Что нужно знать о горутинах и каналах

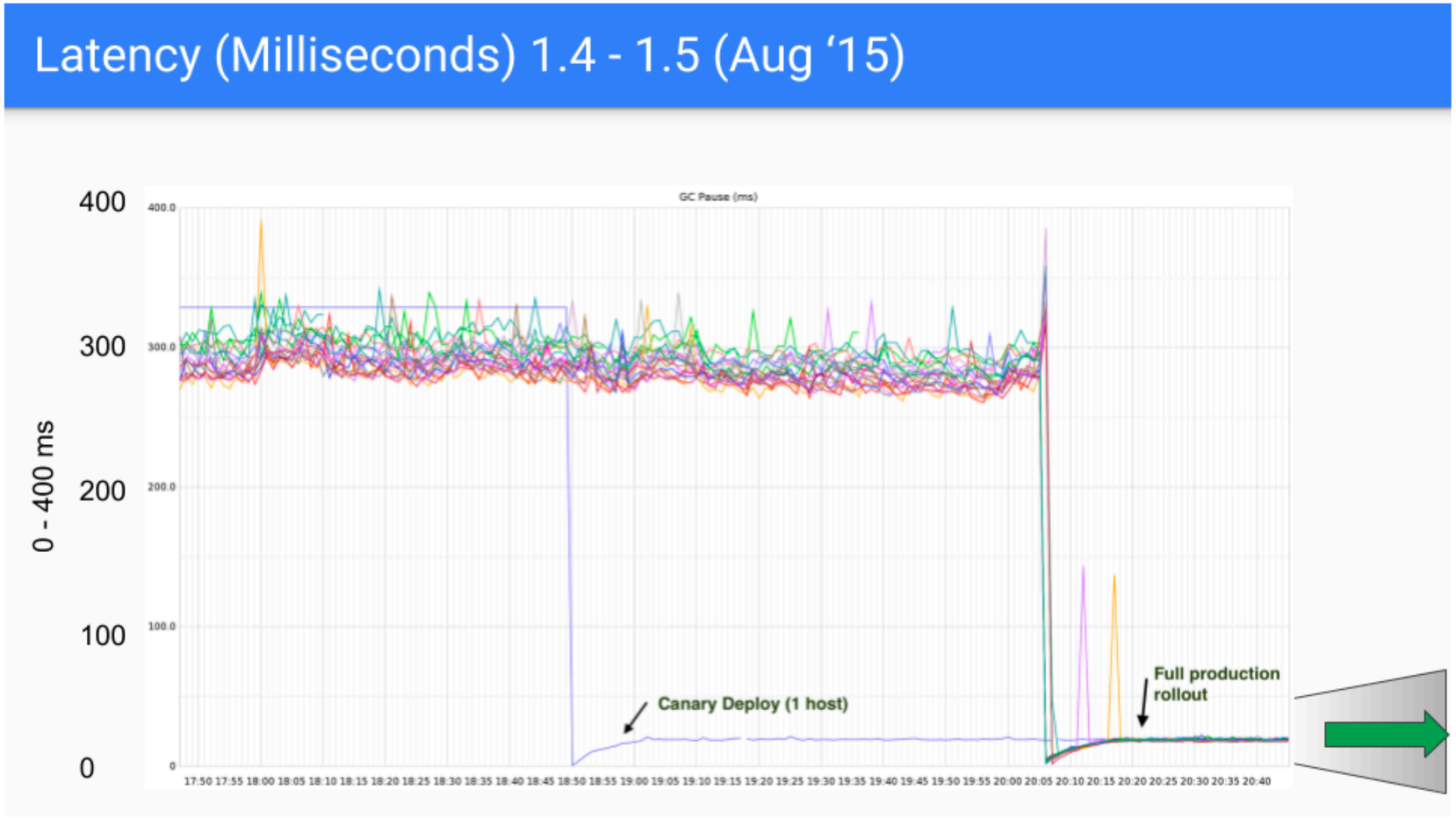
- 1 горутина ~ 4.5 kB памяти. 1 GB — больше 200 тысяч горутин
- переключения между горутинами происходят только в определенных местах кода (кооперативная многозадачность)
- блокирующий системный вызов — одно из таких мест
- при переключении используется шедулер Go runtime, который дешевле системного
- горутины могут выполняться на разных ядрах
- другое место переключения — блокировка на записи/чтении канала. Каналы широко используются для синхронизации
- канал — кольцевой буфер. Производительность канала — около 10 M сообщений в секунду. Пересылка занимает 90 — 250 ns

Горутины, процессоры и машины

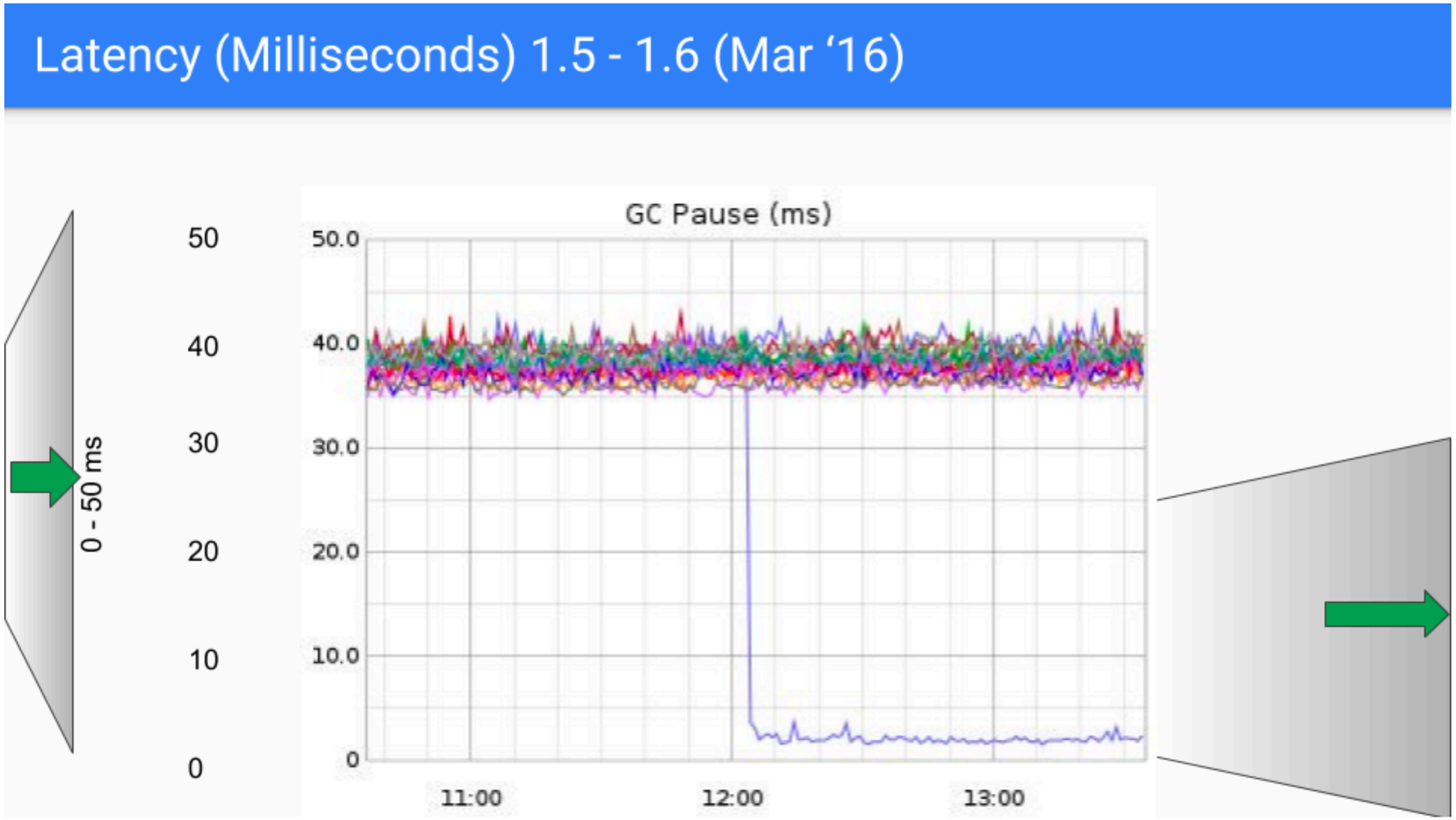


но сборщик мусора...

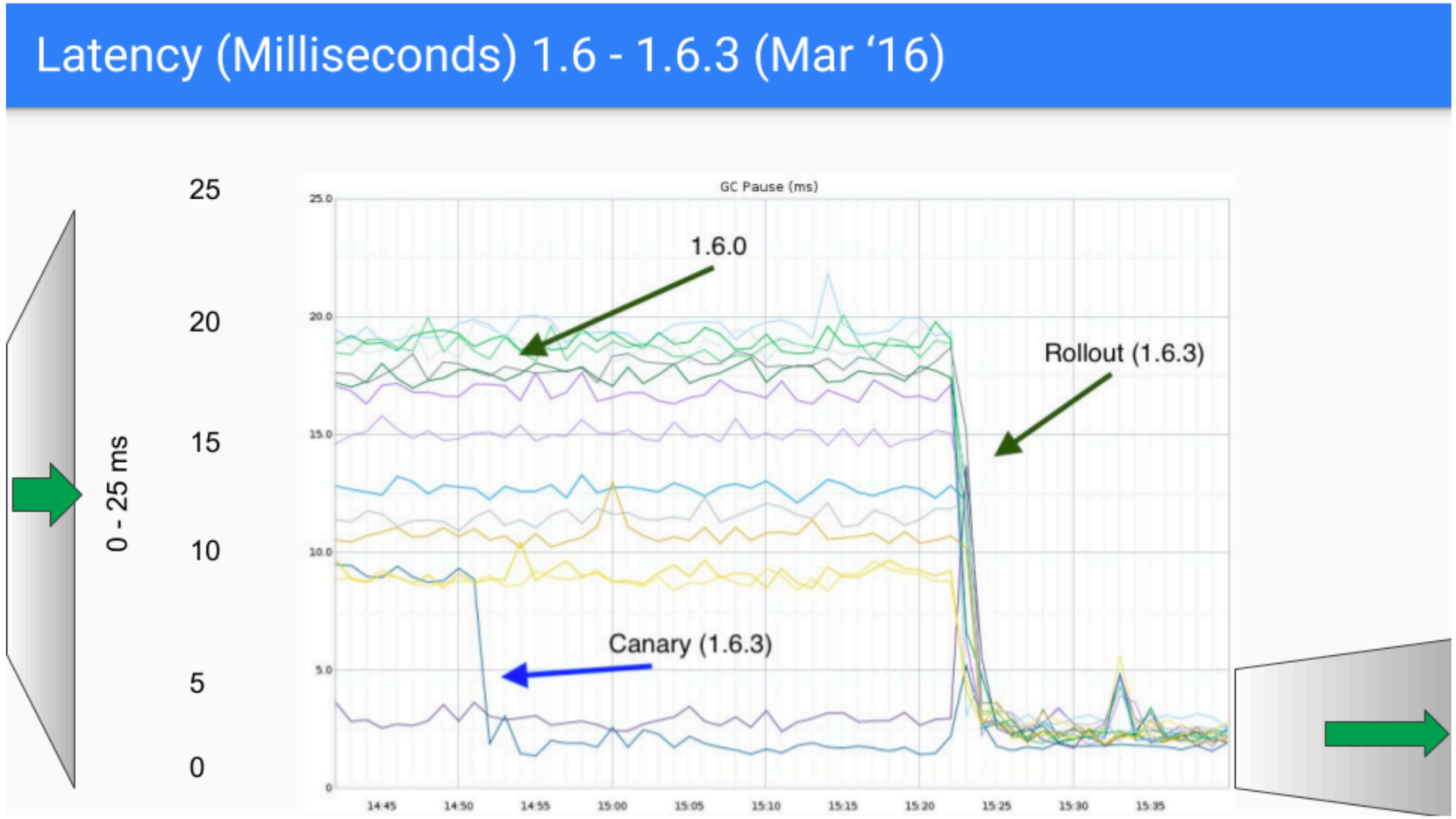
...не такая уж и проблема



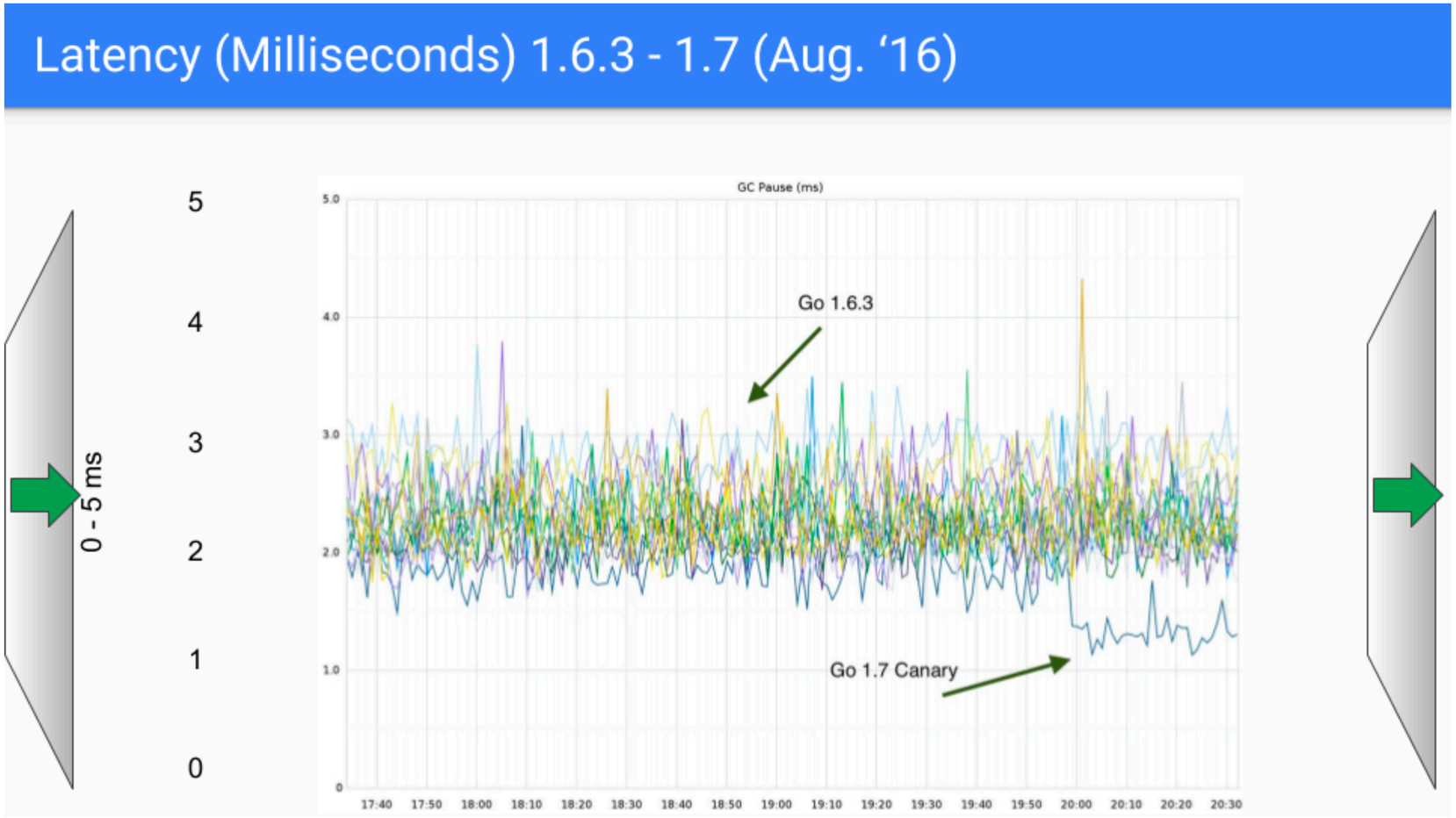
...а можно и лучше



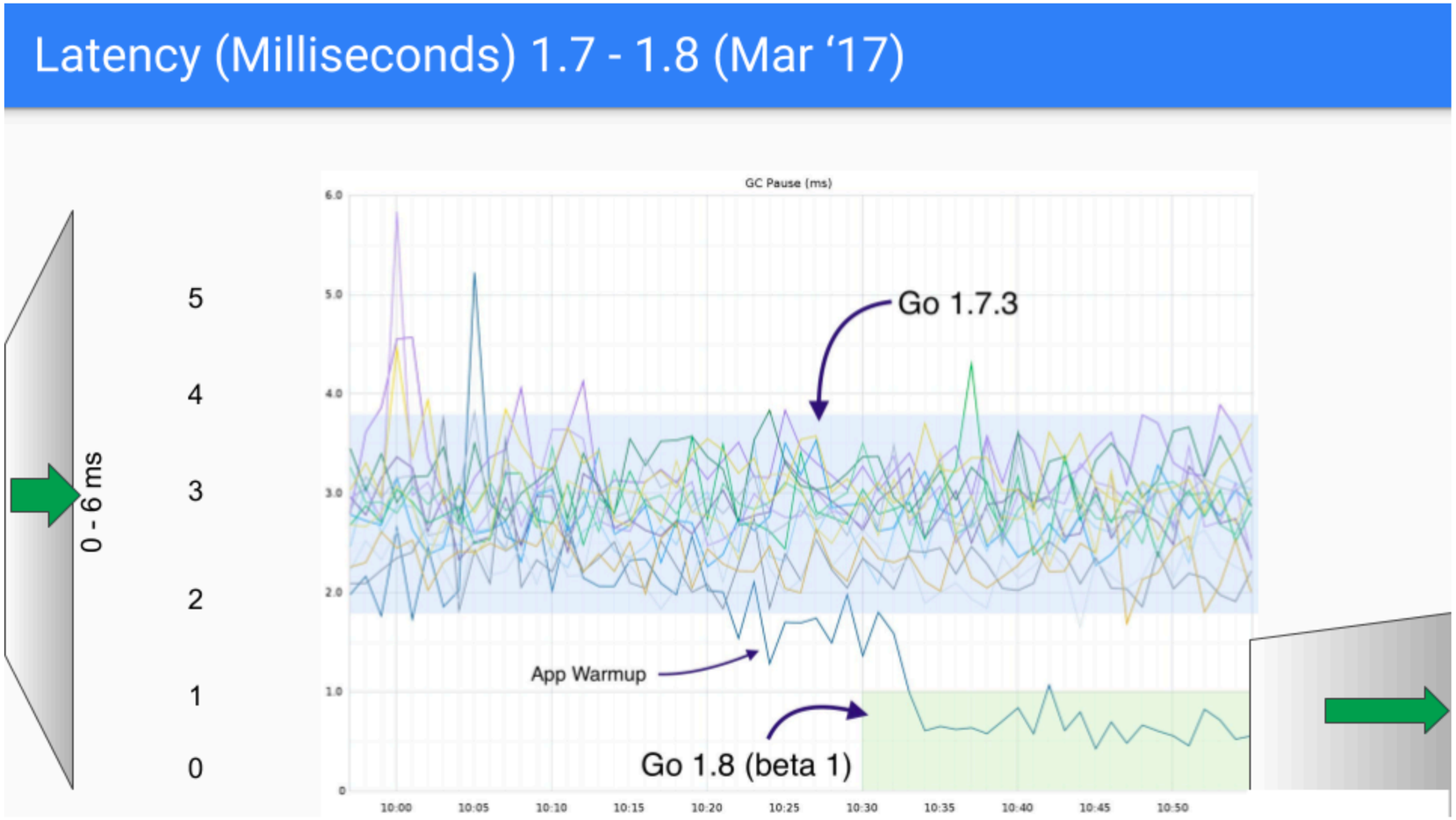
...и еще лучше


Latency (Milliseconds) 1.6 - 1.6.3 (Mar '16)
0 - 25 ms
25
20
15
10
5
0
GC Pause (ms)
1.6.0
Rollout (1.6.3)
Canary (1.6.3)
25.0
20.0
15.0
10.0
5.0
0
14:45
14:50
14:55
15:00
15:05
15:10
15:15
15:20
15:25
15:30
15:35

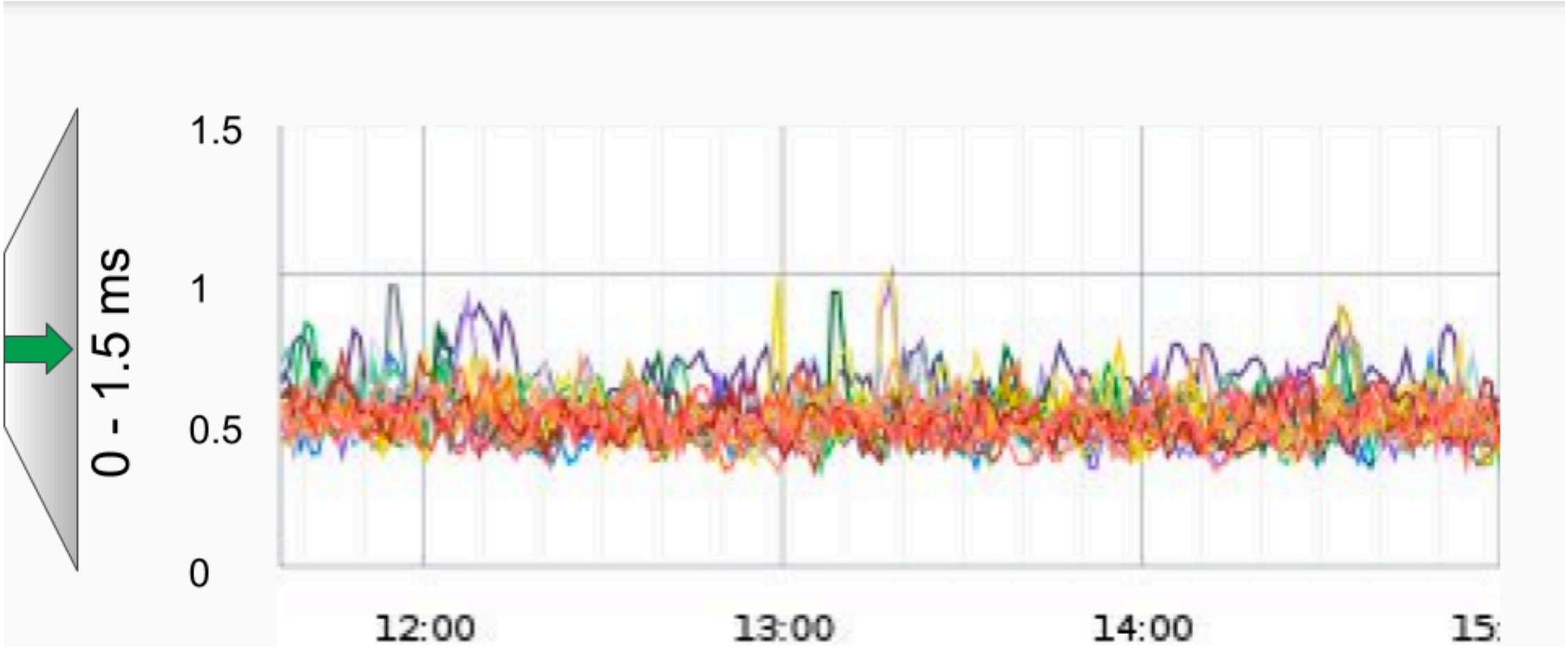
...и еще!


Latency (Milliseconds) 1.6.3 - 1.7 (Aug. ‘16)
0 - 5 ms
5
4
3
2
1
0
GC Pause (ms)
5.0
4.0
3.0
2.0
1.0
0
Go 1.6.3
Go 1.7 Canary
17:40
17:50
18:00
18:10
18:20
18:30
18:40
18:50
19:00
19:10
19:20
19:30
19:40
19:50
20:00
20:10
20:20
20:30

...и еще!!


Latency (Milliseconds) 1.7 - 1.8 (Mar '17)
0 - 6 ms
5
4
3
2
1
0
GC Pause (ms)
6.0
5.0
4.0
3.0
2.0
1.0
0
Go 1.7.3
App Warmup
Go 1.8 (beta 1)
10:00
10:05
10:10
10:15
10:20
10:25
10:30
10:35
10:40
10:45
10:50

...И ВОТ



на практике GC
мешает нечасто. Но помнить
о его наличии стоит

Как писать сценарии?

- интерпретатор Python (медленно, сыро)
- транспилятор Python в Go (сыро, не все поддерживается)
- интерпретатор JS (сыро, медленно, другой язык)
- **большая вероятность наступить на разложенные грабли**

Но что, если...

идея: собираем пушку перед стрельбой

компиляция пушки на Go занимает всего несколько секунд

Преимущества реализации пушки на Go:

- легко достичь параллелизма с помощью горутин
- легко пересылать данные между горутинами с помощью каналов
- статическая типизация: меньше вероятность ошибок
- возможность управлять памятью
- пушка собирается в один бинарь со всеми зависимостями
- кроссплатформенность и кросс-компиляция

это все работает и для пользовательских сценариев, причем внутри сценария тоже можно использовать горутины и каналы

Pandora: опыт Яндекса

Для чего использовали:

- высокопроизводительные HTTP сценарии
- HTTP/2
- Postgress
- protobuf

Сложность внедрения:

- разобраться с нуля — пара дней при наличии прототипа
- 2 — 7 дней на переезд готового теста с JMeter
- код гораздо проще отдать разработчикам, чем GUI сценарий (известен всего один случай в нашей истории =))

Производительность Pandora

Производительность "из коробки", в nginx, по 32 ядра, 1 Гбит/С:

- **23 Krps** HTTP Connection: Close
- **95 Krps** HTTP Connection: Keep-Alive
- **35 Krps** сценарии

yandexpandora.readthedocs.io/en/develop/performance.html

Phantom, JMeter, BFG и Pandora

Пушки для Yandex.Tank (из коробки):

- Phantom: быстрый, нет сценариев, только простые протоколы
- JMeter: средняя скорость, сценарии в GUI или Groovy, протоколы реализованы в плагинах
- BFG: самая медленная, сценарии на Python, протоколы в Python библиотеках
- **Pandora**: почти такая же быстрая, как Phantom, сценарии на Go, протоколы в библиотеках

Немного истории

- 2015: первые стрельбы в SPDY и HTTP/2 простым скриптом на Go
- обобщение до стрелялки с поддержкой альтернативных пушек
- затем первый рефакторинг от **m0sth8**
- сравниваемся с фантомом: “чуть хуже”
- 2017: **skipor** присоединился к проекту и очень многое переделал
- 2018: мы осознали, что можно писать на Go сценарии. Доработка Pandora для поддержки пользовательских сценариев. Первые осторожные эксперименты
- 2019: используется для стрельб
- а сегодня — первый публичный рассказ

Пробуем стрелять

Первые шаги

Скачайте бинарь: github.com/yandex/pandora/releases/tag/0.3.0

> **стрельба в стиле Phantom: посылаем последовательно запросы из файла по протоколу HTTP**

Нам нужно два файла:

- pandora.yaml — файл конфигурации
- ammo.txt — файл с патронами

Если стреляем с Яндекс.Танком, то еще один:

- load.yaml — файл конфигурации Танка

pandora.yaml

```yaml
pools:
  - id: HTTP pool                    # название пула (пулов может быть несколько)
    gun:
      type: http                     # тип пушки
      target: example.org:80         # куда стрелять
    ammo:
      type: uri                      # формат патронов
      file: ./ammo.uri               # файл с патронами
    result:
      type: phout                    # формат отчета
      destination: ./phout.log       # файл отчета
```

```yaml
pools:
  - id: HTTP pool                    # название пула (пулов может быть несколько)
    gun:
      type: http                     # тип пушки
      target: example.org:80         # куда стрелять
    ammo:
      type: uri                      # формат патронов
      file: ./ammo.uri               # файл с патронами
    result:
      type: phout                    # формат отчета
      destination: ./phout.log       # файл отчета
    rps:                             # расписание стрельбы
      type: line                     # линейный рост
      from: 1                        # от 1 RPS
      to: 100000                     # до 100000 RPS
      duration: 60s                  # за 60 секунд

    startup:                         # расписание старта инстансов
      type: once                     # запустить 10000 инстансов за раз
      times: 10000
```

ammo.uri

```
/my/first/url
/my/second/url?param=1&other_param=2
/my/third/url
```

Запускаем

```
direvius@tank05y:~/tests$ pandora ./pandora.yaml
2019-03-27T14:15:08.921+0300  INFO cli/cli.go:185  Reading config  {"file": "./pandora.yaml"}
2019-03-27T14:15:08.925+0300  INFO engine/engine.go:137 Pool run started{"pool": "HTTP pool"}
2019-03-27T14:15:09.944+0300  INFO cli/expvar.go:40[ENGINE] 32 resp/s; 44 req/s; 62732 users; 12 active

2019-03-27T14:15:10.925+0300  INFO cli/expvar.go:40[ENGINE] 137 resp/s; 125 req/s; 100000 users; 0 active

2019-03-27T14:15:11.925+0300  INFO cli/expvar.go:40[ENGINE] 209 resp/s; 209 req/s; 100000 users; 0 active

2019-03-27T14:15:12.925+0300  INFO cli/expvar.go:40[ENGINE] 293 resp/s; 293 req/s; 100000 users; 0 active

2019-03-27T14:15:13.925+0300  INFO cli/expvar.go:40[ENGINE] 375 resp/s; 375 req/s; 100000 users; 0 active
```

Запускаем

```
direvius@tank05y:~/tests$ pandora ./pandora.yaml
2019-03-27T14:15:08.921+0300  INFO cli/cli.go:185  Reading config  {"file": "./pandora.yaml"}
2019-03-27T14:15:08.925+0300  INFO engine/engine.go:137 Pool run started{"pool": "HTTP pool"}
2019-03-27T14:15:09.944+0300  INFO cli/expvar.go:40[ENGINE] 32 resp/s; 44 req/s; 62732 users; 12 active

2019-03-27T14:15:10.925+0300  INFO cli/expvar.go:40[ENGINE] 137 resp/s; 125 req/s; 100000 users; 0 active

2019-03-27T14:15:11.925+0300  INFO cli/expvar.go:40[ENGINE] 209 resp/s; 209 req/s; 100000 users; 0 active

2019-03-27T14:15:12.925+0300  INFO cli/expvar.go:40[ENGINE] 293 resp/s; 293 req/s; 100000 users; 0 active

2019-03-27T14:15:13.925+0300  INFO cli/expvar.go:40[ENGINE] 375 resp/s; 375 req/s; 100000 users; 0 active
```

Результаты в phout.log

```
1553685308.926  __EMPTY__       5246    0       0       0       0       0       0       0       0       301
1553685309.085  __EMPTY__       40846   0       0       0       0       0       0       0       0       301
1553685309.146  __EMPTY__       67795   0       0       0       0       0       0       0       0       301
1553685309.214  __EMPTY__       35739   0       0       0       0       0       0       0       0       301
```

clck.ru/FS3pC — пример анализа результатов на Python

clck.ru/FS3qw — описание полей

Подключаем танк (load.yaml)

```yaml
phantom:
  enabled: false                    # выключаем phantom
pandora:
  enabled: true                     # включаем pandora
  config_file: pandora.yaml         # конфиг pandora
```

Результаты

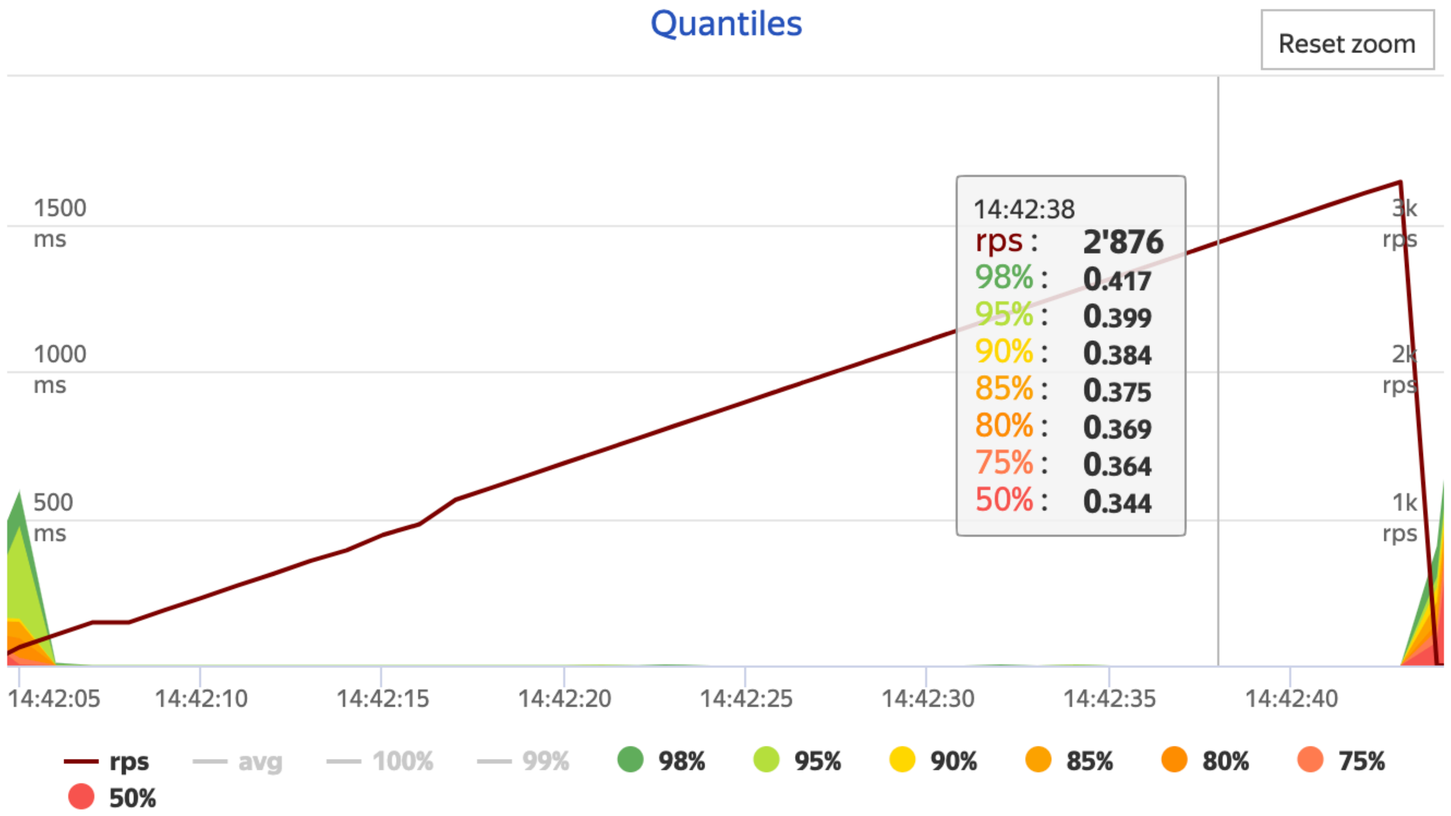
- JSON с агрегированными данными: test_data.log
- мониторинг: monitoring.log

```
direvius@tank05y:~/tests/logs/2019-03-27_14-29-05.502379$ ls
agent_collector_localhost.cfg  agent_startup_localhost.cfg  monitoring.log              phout.log
agent_customs_localhost.cfg    configinitial.yaml           monitoring_u1_7jb.xml       tank.log
agent_localhost.log            finish_status.yaml           pandora_config_YLrHeG.yaml  test_data.log
agent_localhost.rawdata        jobno_file.txt               pandora_jO5hTC.log          validated_conf.yaml
```

Заливаем в Overload (load.yaml)

```yaml
phantom:
  enabled: false                    # выключаем phantom
pandora:
  enabled: true                     # включаем pandora
  config_file: pandora.yaml         # конфиг pandora
overload:
  enabled: true
  token_file: token.txt
```

Результаты

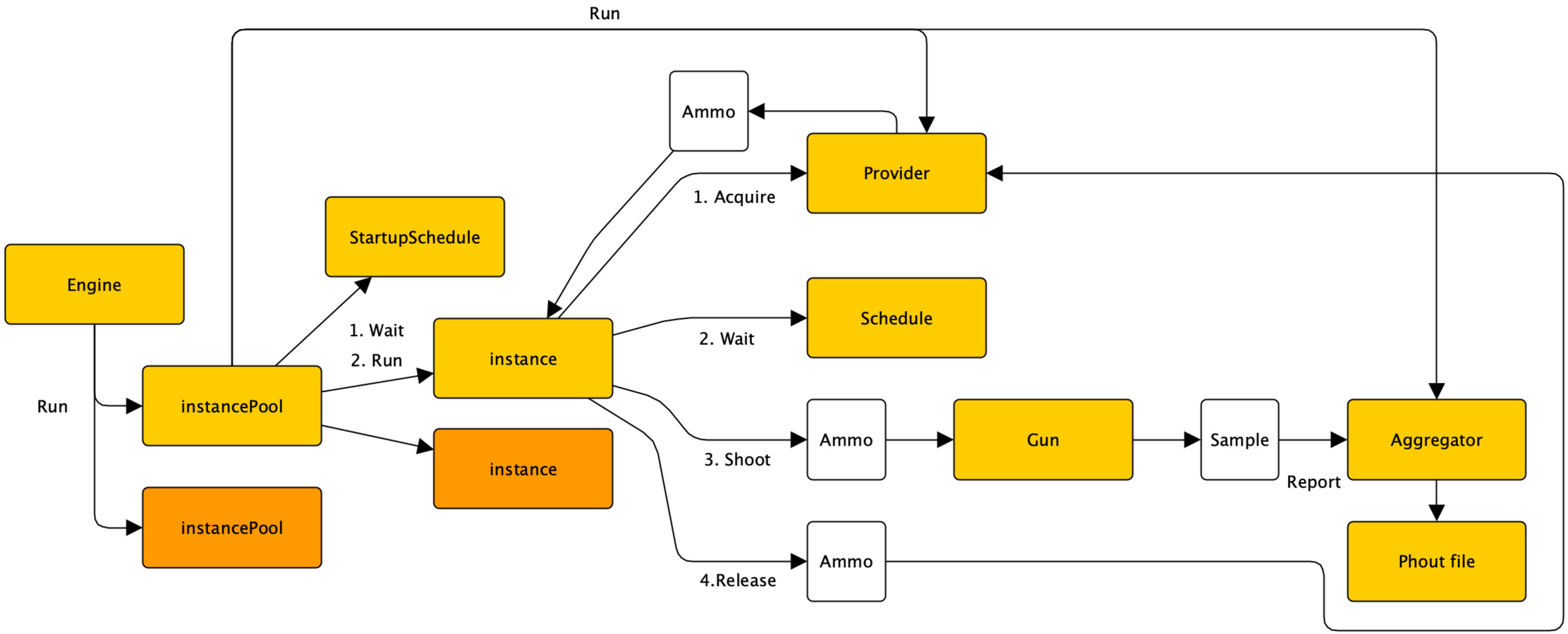


Графики, графики

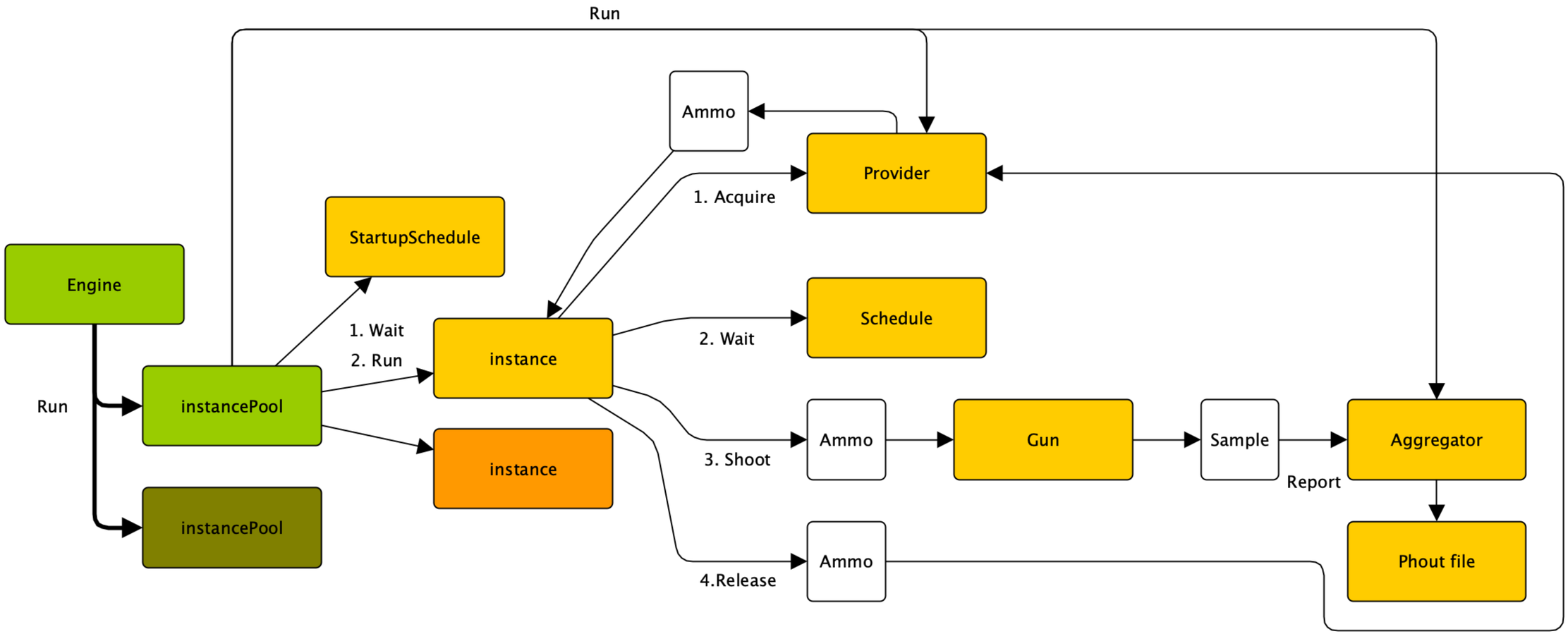
overload.yandex.net/172420

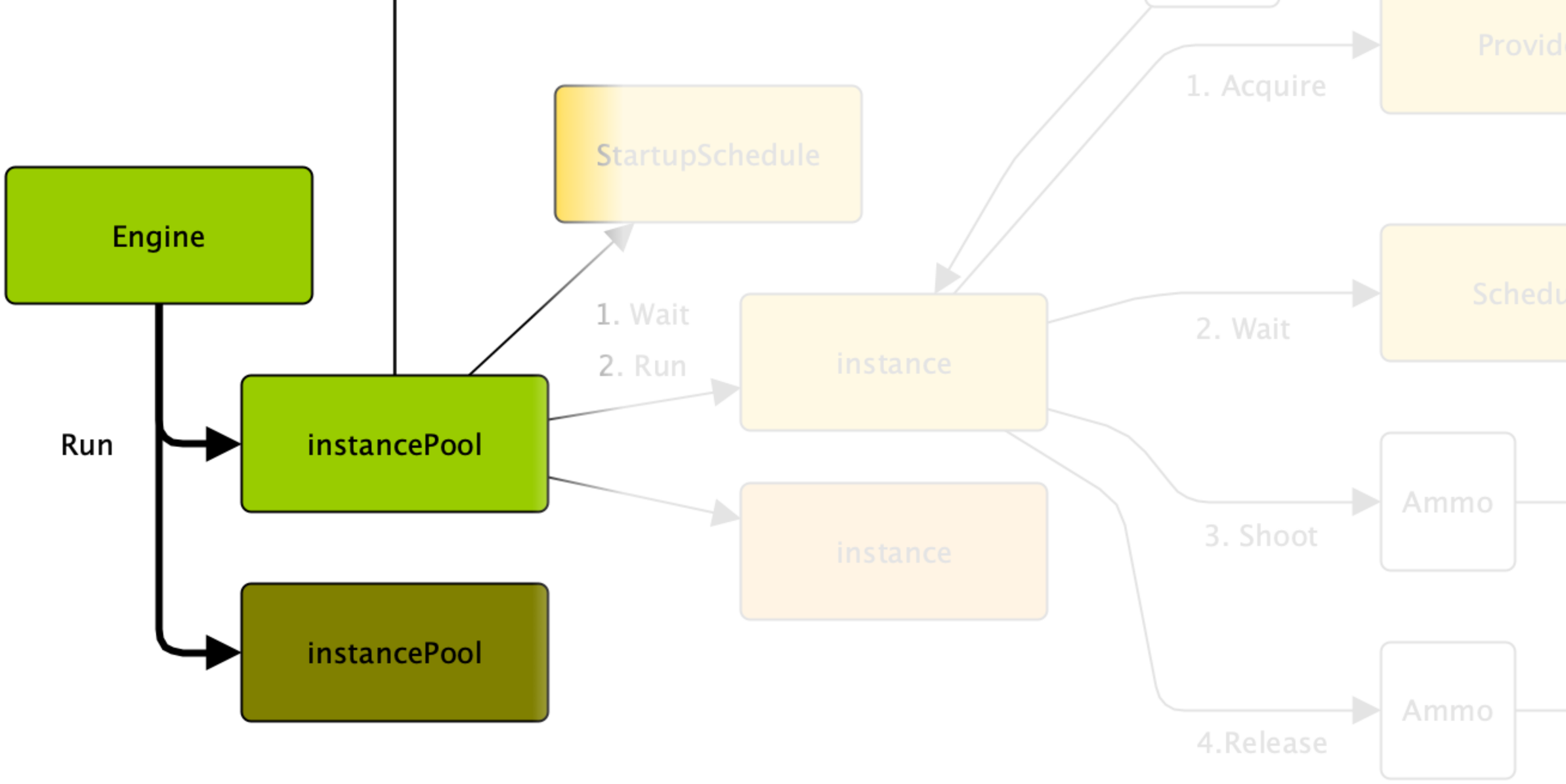
Архитектура

Архитектура Pandora



Instance Pool



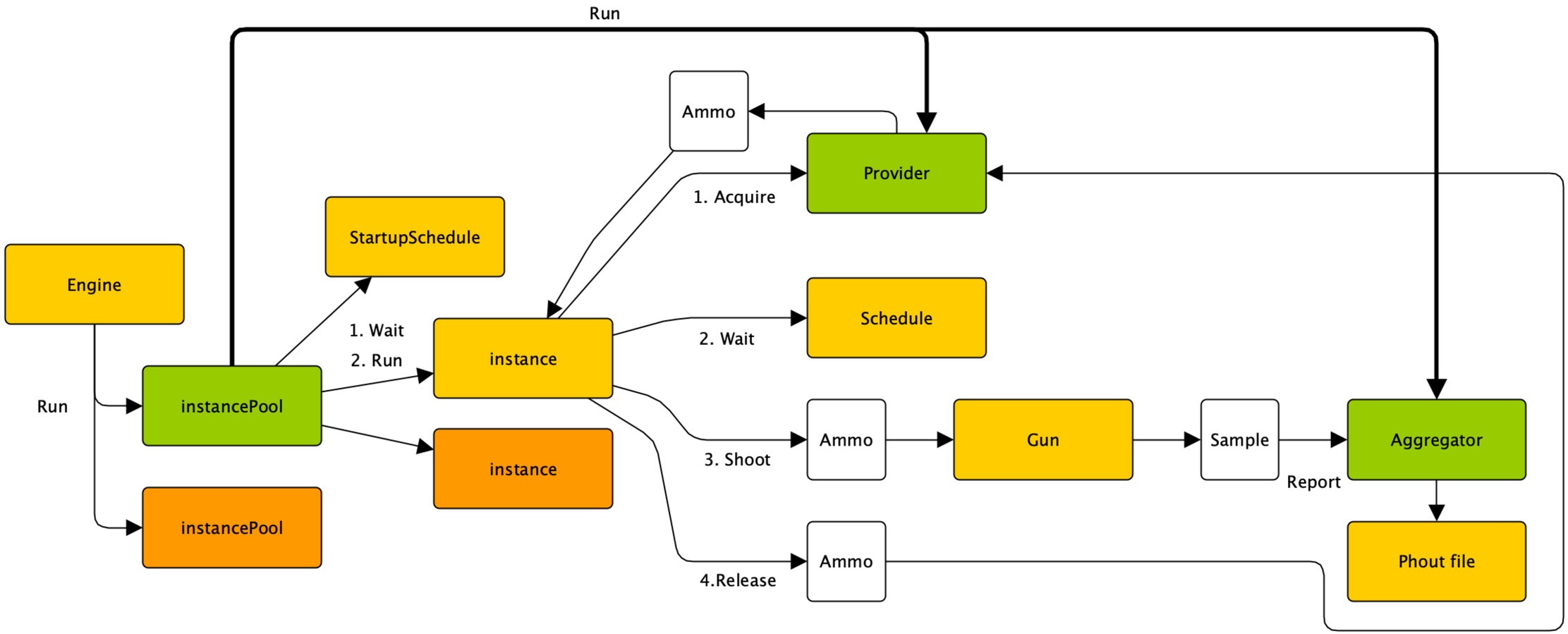

Engine
Run
instancePool
instancePool
StartupSchedule
1. Wait
2. Run
instance
instance
1. Acquire
2. Wait
3. Shoot
4.Release
Ammo
Ammo

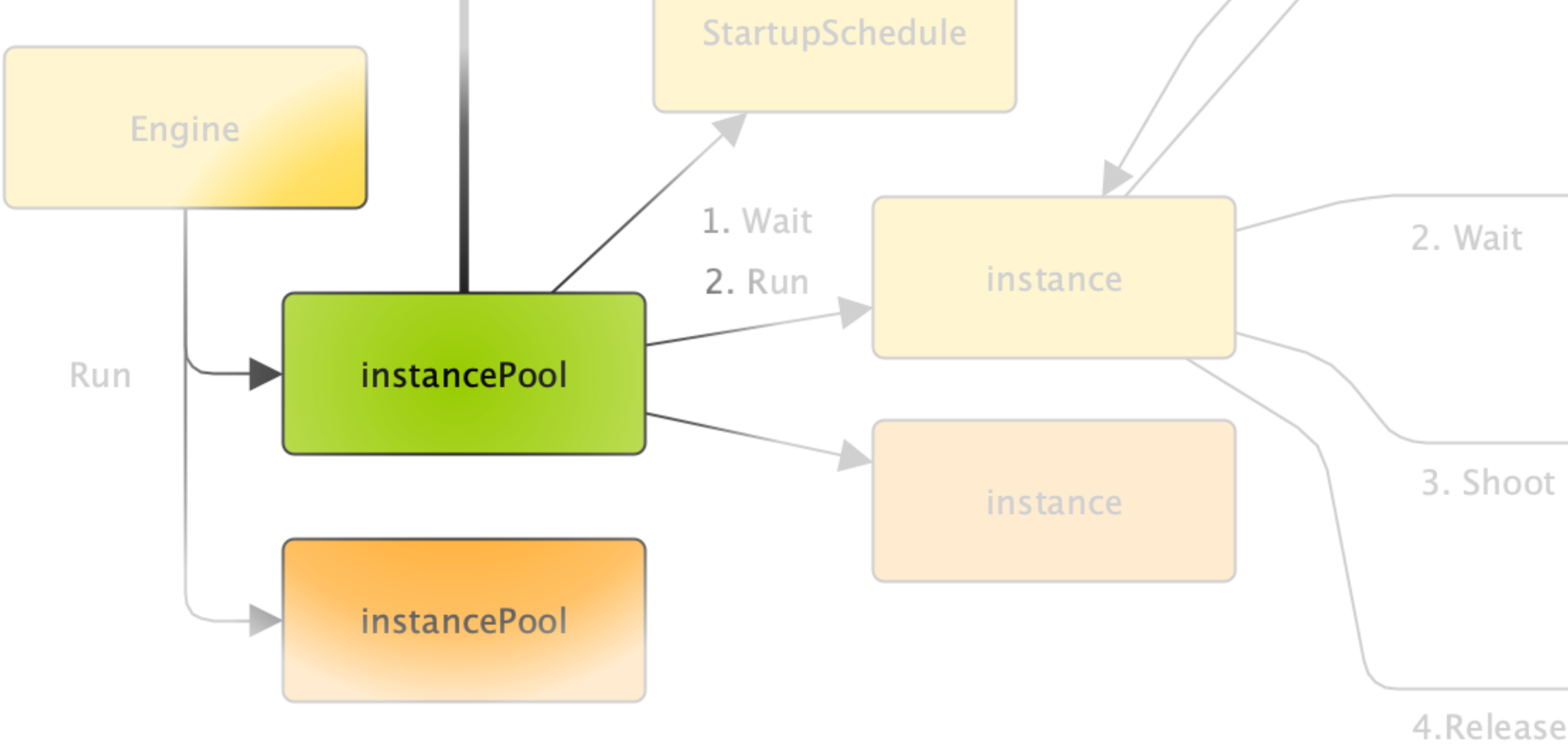
Пулы виртуальных пользователей

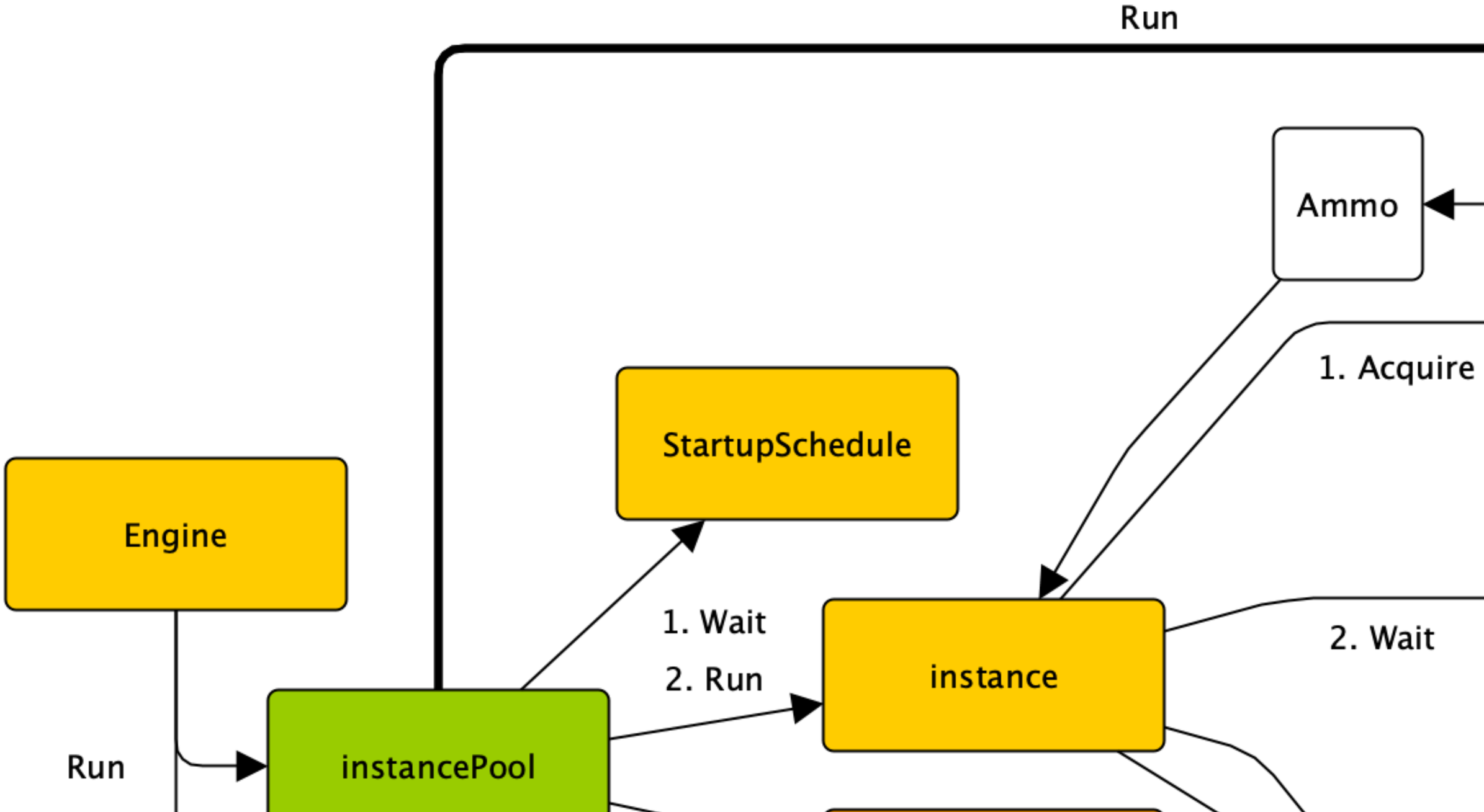
- у каждого пула свое расписание, патроны, мишени
- можно использовать разные пушки
- отдельный вывод результатов

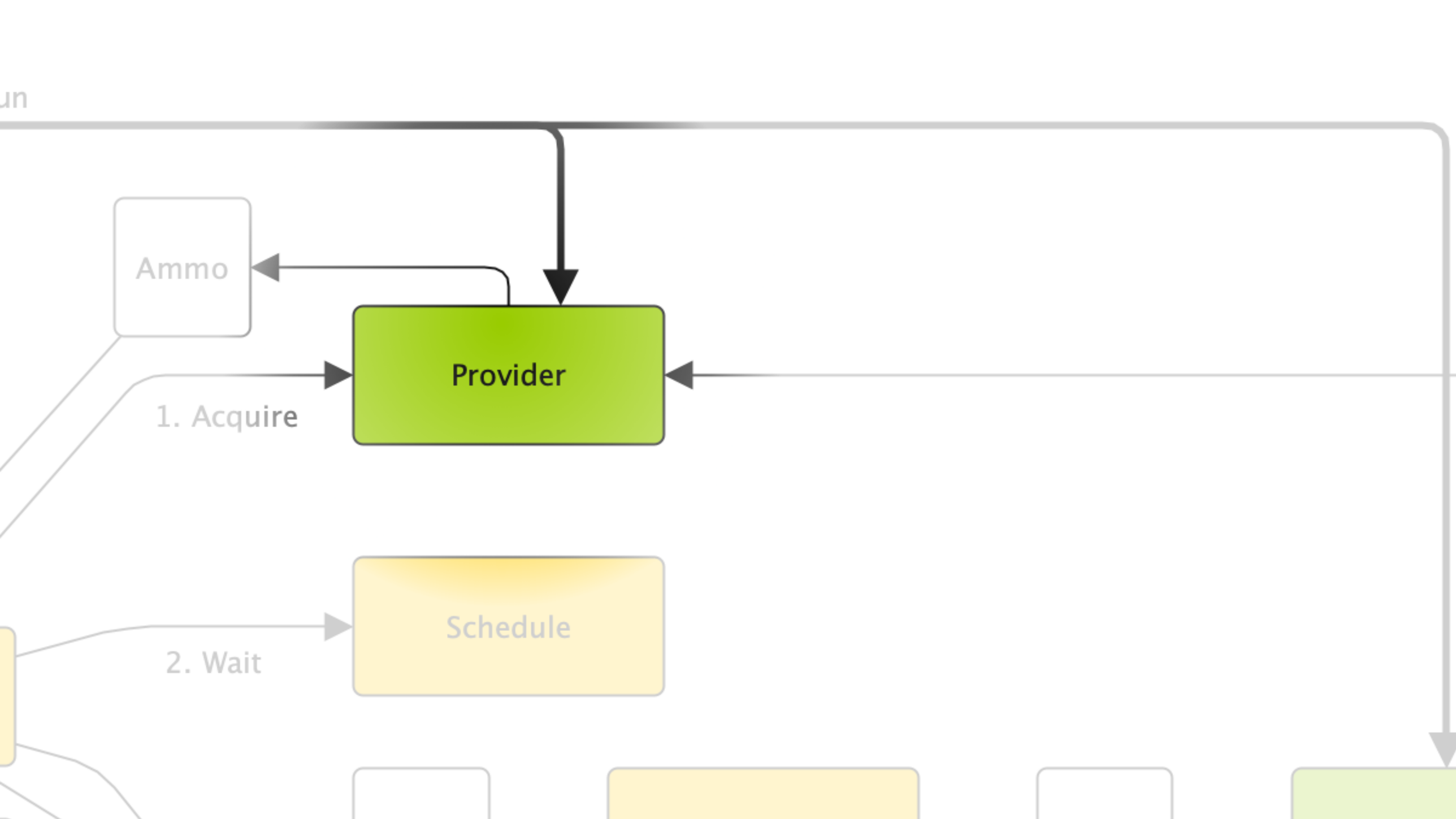
```
pools:
  - id: HTTP pool                    # название пула (пулов может быть несколько)
    gun:
      type: http                     # тип пушки
      target: example.org:80         # куда стрелять
    ammo:
      type: uri                      # формат патронов
      file: ./ammo.uri               # файл с патронами
    result:
      type: phout                    # формат отчета
      destination: ./phout.log       # файл отчета
    rps:                             # расписание стрельбы
      type: line                     # линейный рост
      from: 1                        # от 1 RPS
      to: 100000                     # до 100000 RPS
```

Инициализация Instance Pool




StartupSchedule
Engine
1. Wait
2. Run
instance
2. Wait
Run
instancePool
instance
3. Shoot
instancePool
4.Release

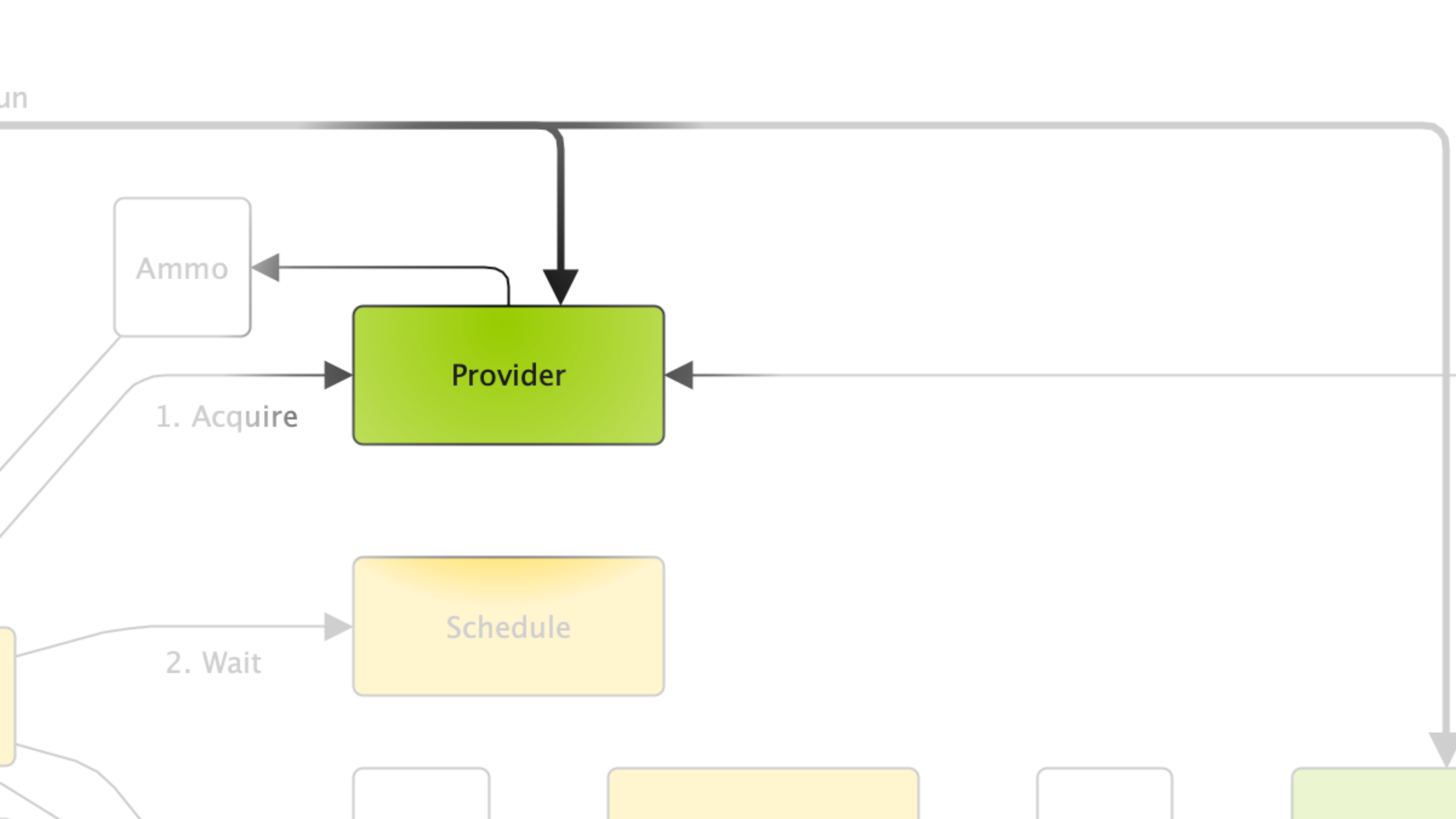

Run
Ammo
1. Acquire
StartupSchedule
Engine
1. Wait
2. Run
instance
2. Wait
Run
instancePool

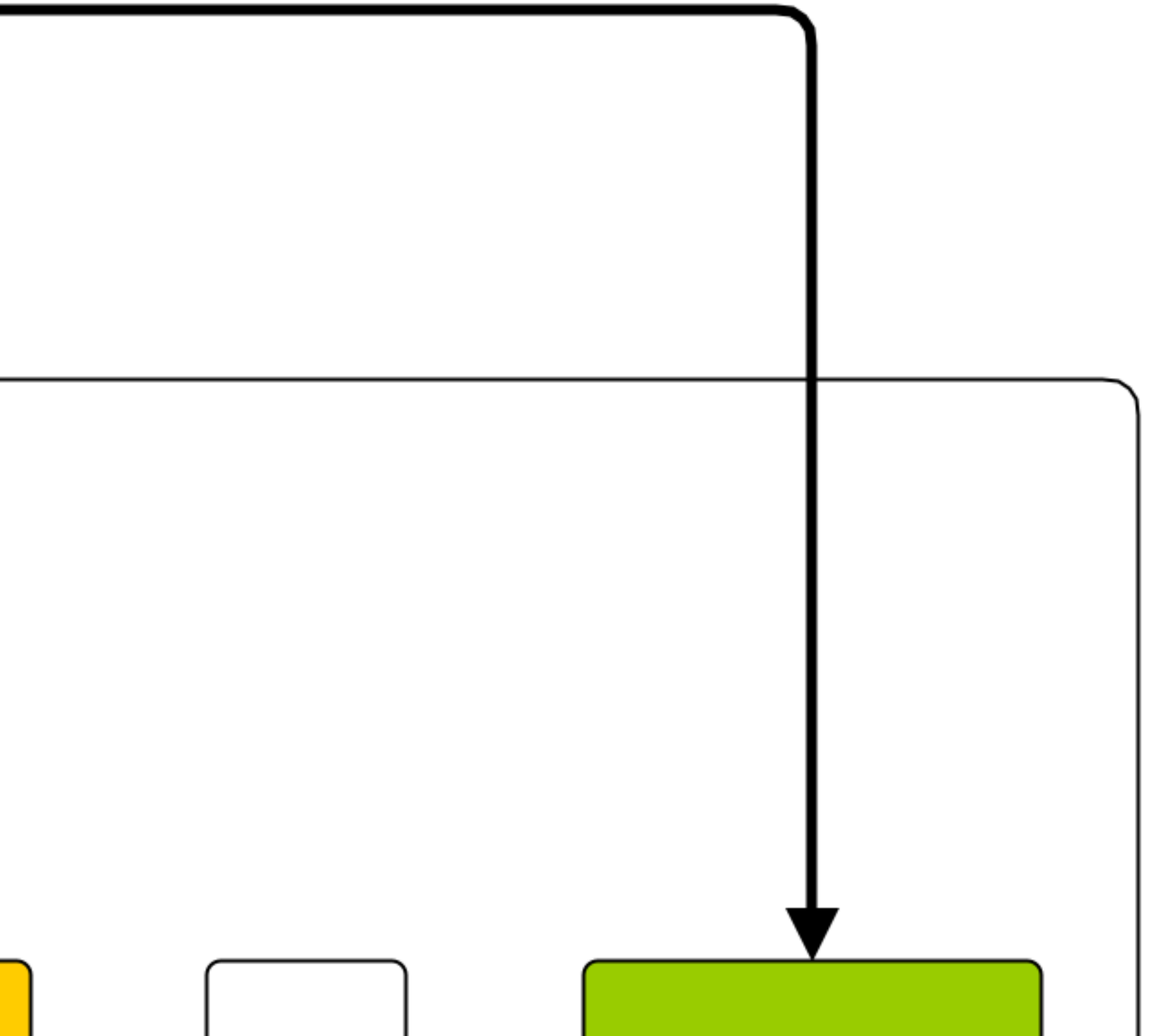

un
Ammo
Provider
1. Acquire
Schedule
2. Wait

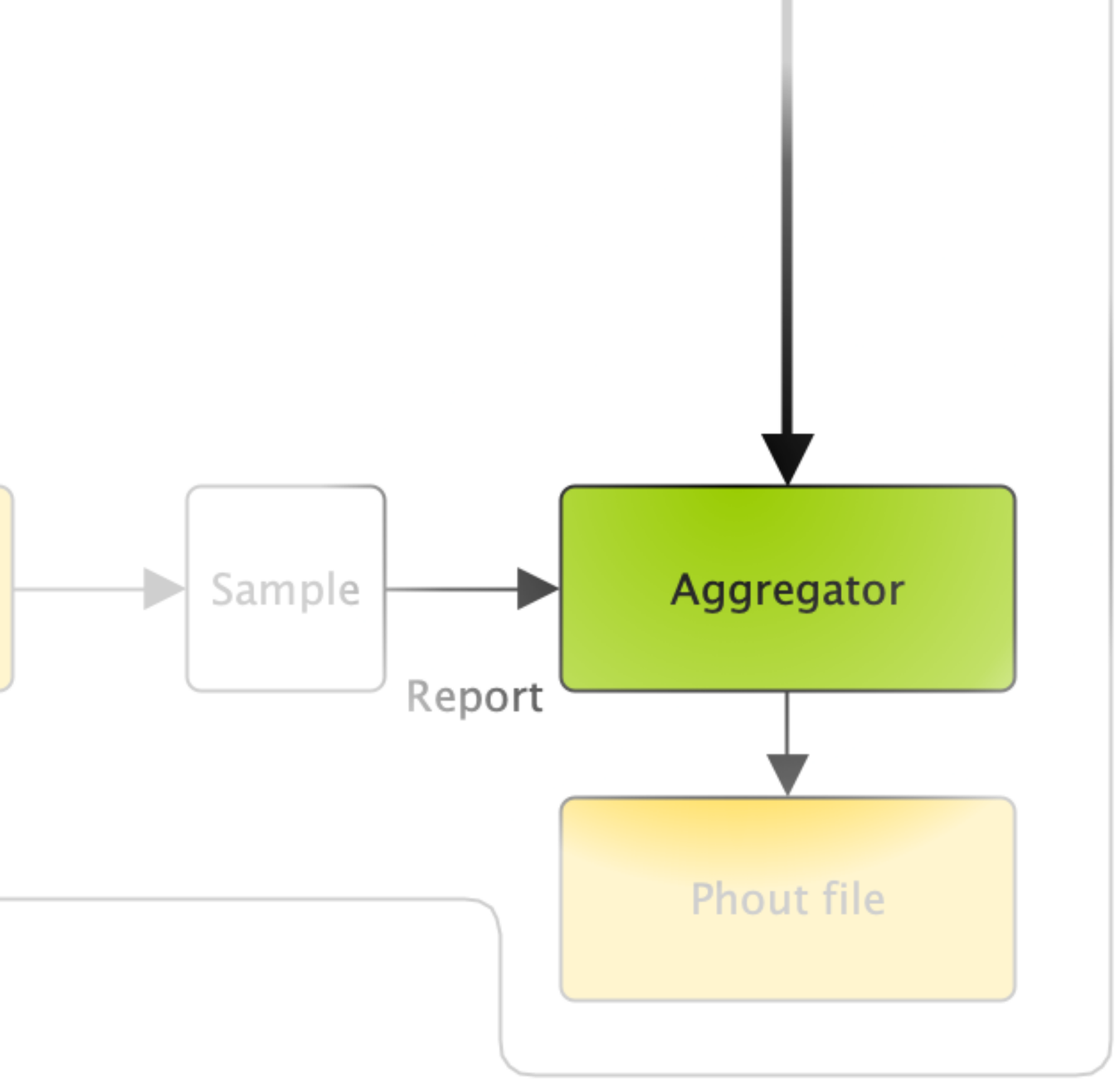
Источник патронов

- патроны — это данные для параметризации запросов
- могут быть разного формата

```
pools:
  - id: HTTP pool                    # название пула (пулов может быть несколько)
    gun:
      type: http                     # тип пушки
      target: example.org:80         # куда стрелять
    ammo:
      type: uri                      # формат патронов
      file: ./ammo.uri               # файл с патронами
    result:
      type: phout                    # формат отчета
      destination: ./phout.log       # файл отчета
    rps:                             # расписание стрельбы
      type: line                     # линейный рост
      from: 1                        # от 1 RPS
      to: 100000                     # до 100000 RPS
```


un
Ammo
Provider
1. Acquire
Schedule
2. Wait



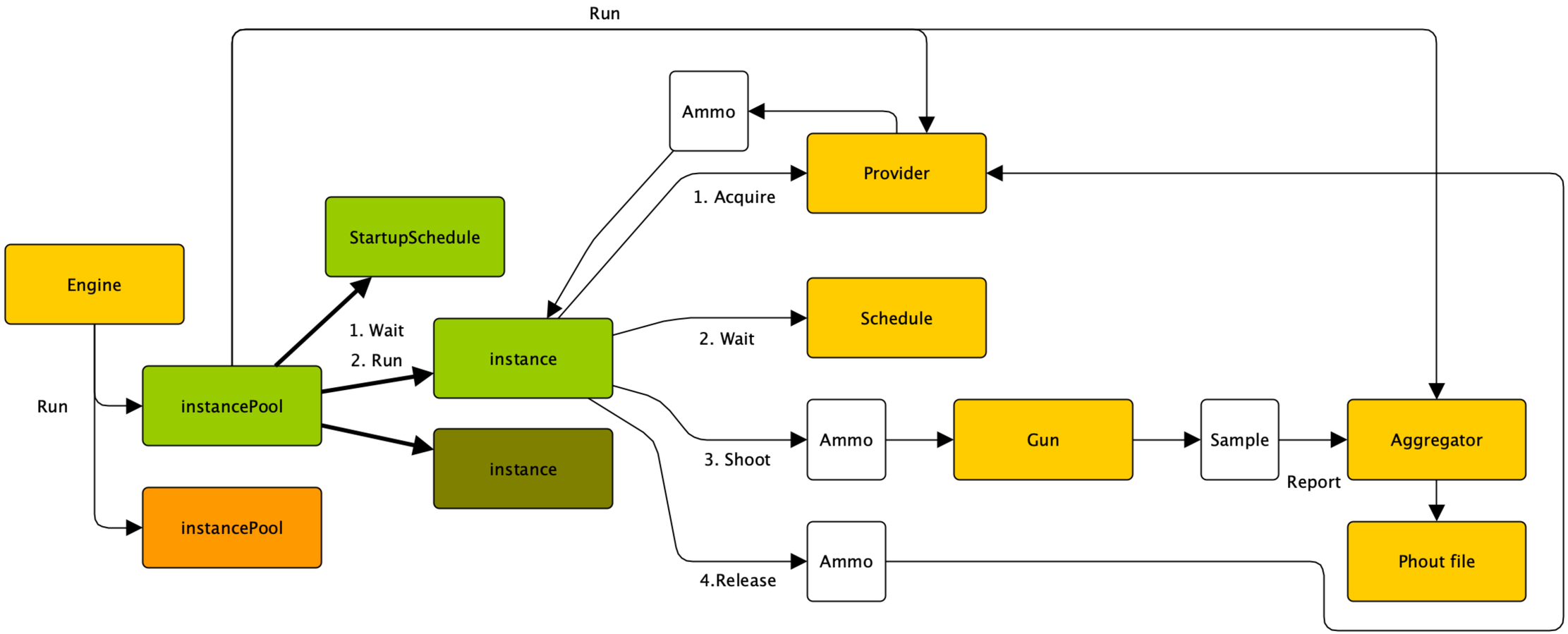

Sample
Aggregator
Report
Phout file

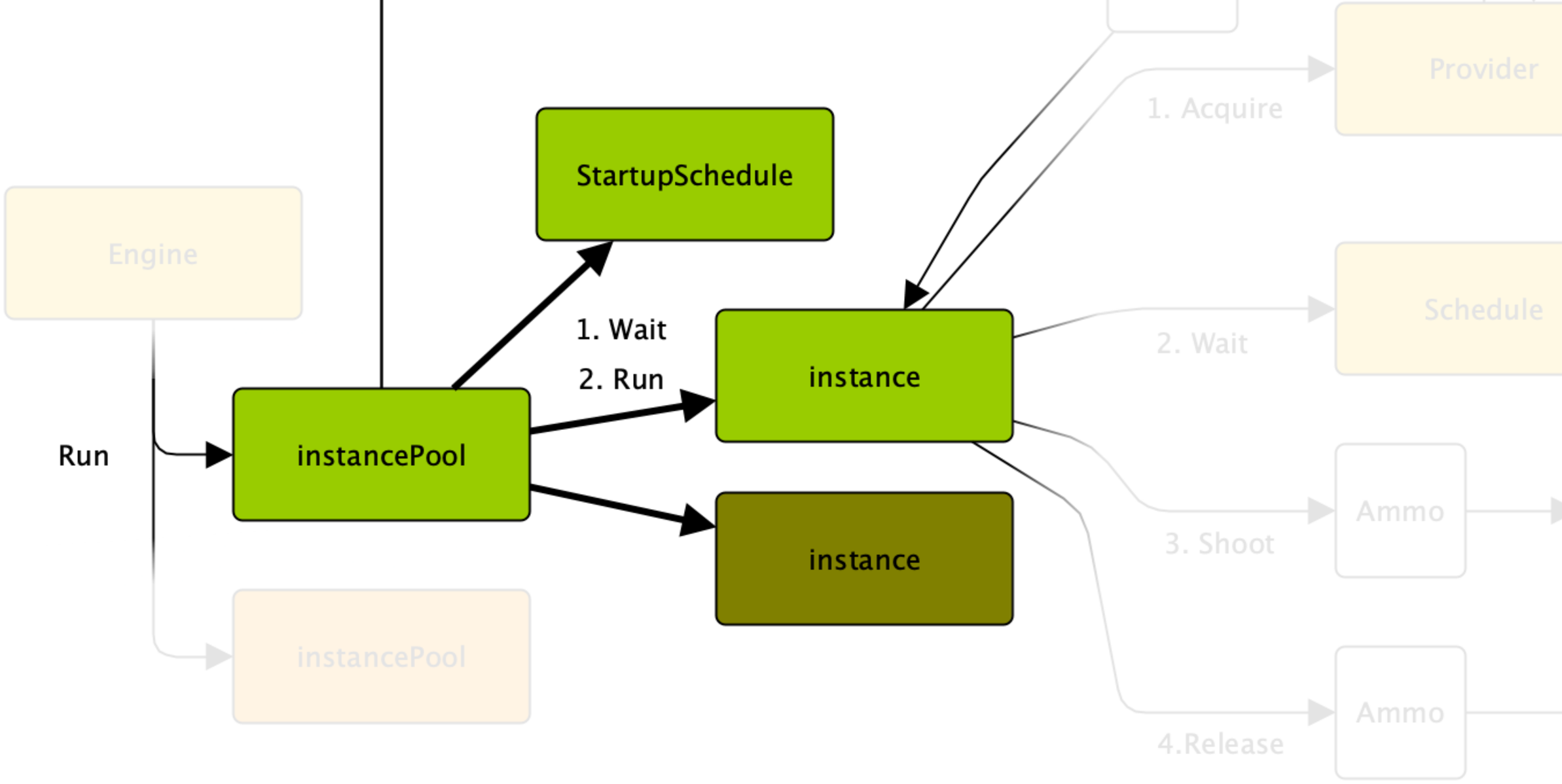
Потребитель результатов

- куда записать результаты обстрела
- файл или база данных, в сыром или обработанном виде

```
pools:
  - id: HTTP pool                    # название пула (пулов может быть несколько)
    gun:
      type: http                     # тип пушки
      target: example.org:80         # куда стрелять
    ammo:
      type: uri                      # формат патронов
      file: ./ammo.uri               # файл с патронами
    result:
      type: phout                    # формат отчета
      destination: ./phout.log       # файл отчета
    rps:                             # расписание стрельбы
      type: line                     # линейный рост
      from: 1                        # от 1 RPS
      to: 100000                     # до 100000 RPS
```

Запуск виртуальных пользователей




StartupSchedule
Engine
1. Wait
2. Run
instance
Run
instancePool
instance
instancePool
Provider
1. Acquire
Schedule
2. Wait
Ammo
3. Shoot
Ammo
4.Release

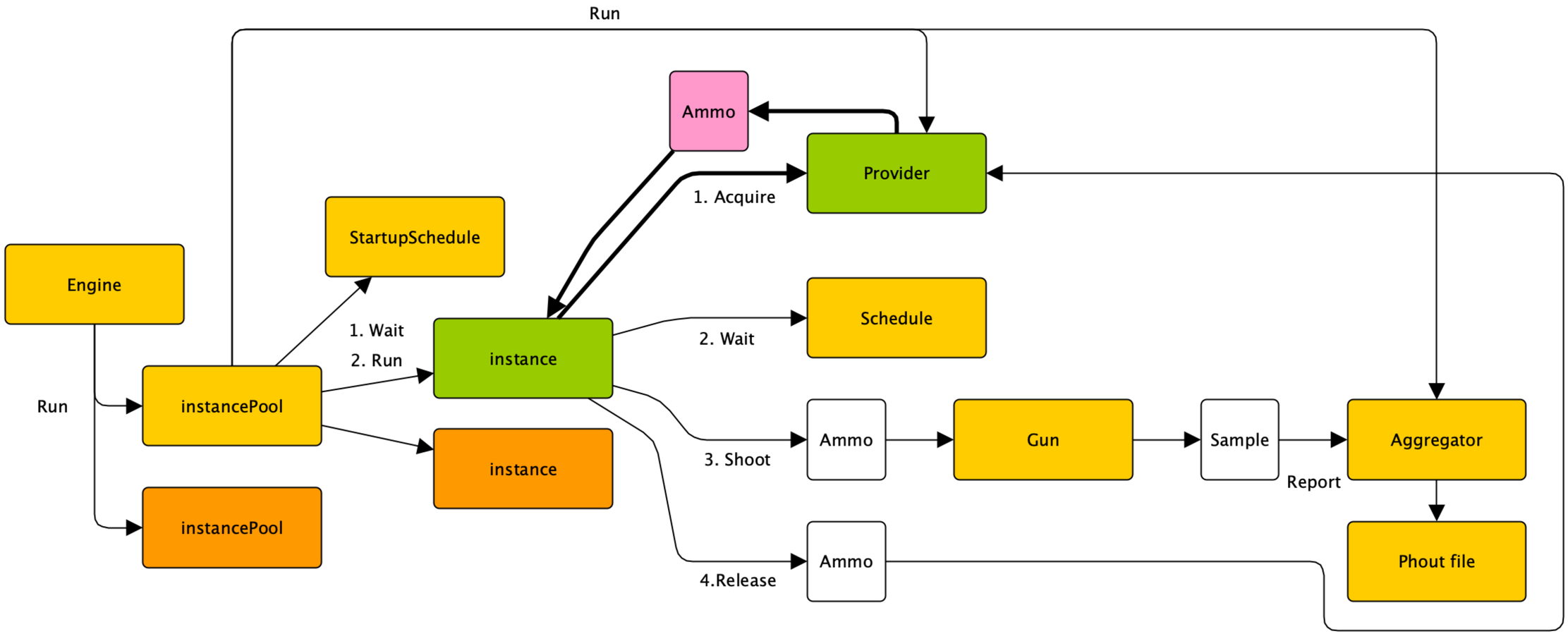
Расписание старта виртуальных пользователей

- можно запустить всех сразу, или увеличивать их количество постепенно

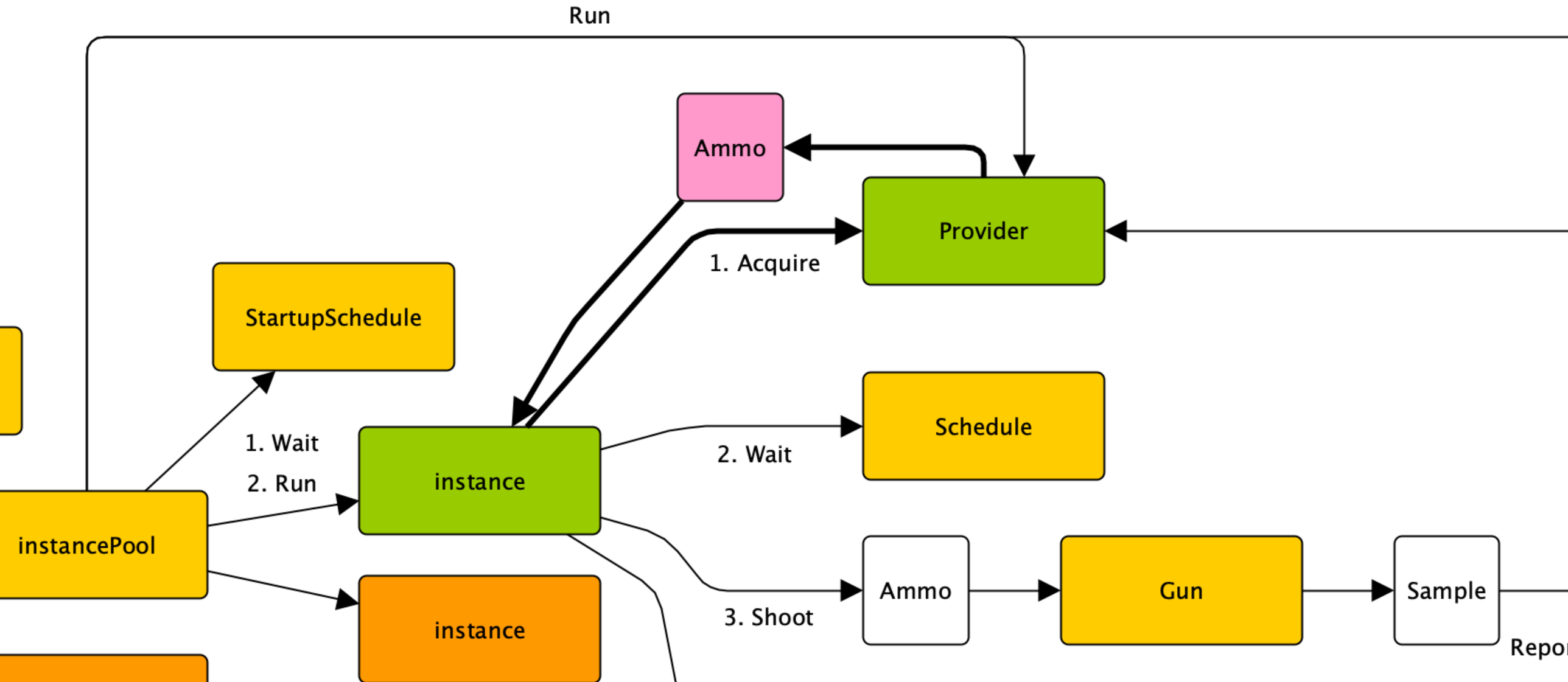
```
pools:
  - ...
    rps:                                # расписание стрельбы
      type: line                        # линейный рост
      from: 1                           # от 1 RPS
      to: 100000                        # до 100000 RPS
      duration: 60s                     # за 60 секунд

    startup:                            # расписание старта инстансов
      type: once                        # запустить 10000 инстансов за раз
      times: 10000
```

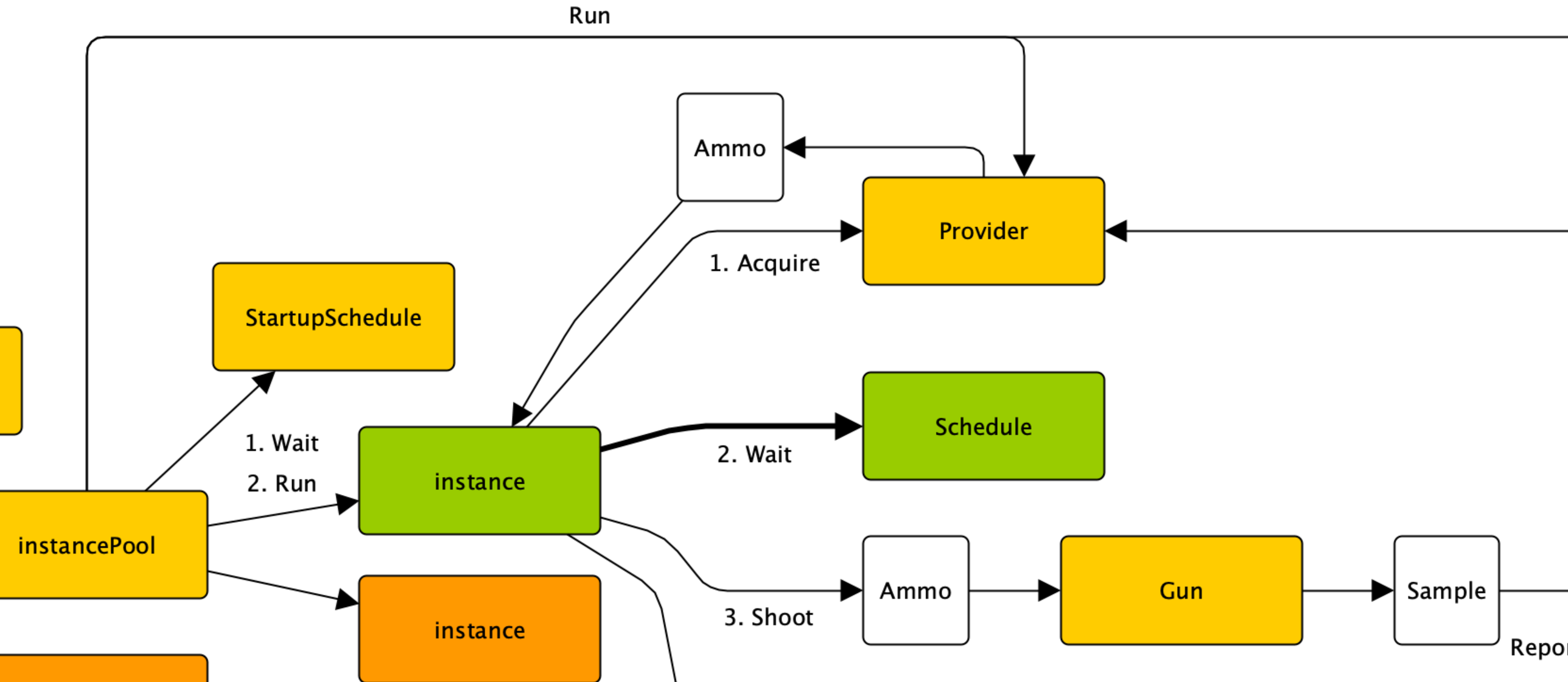
Цикл стрельбы: заряжаем патрон



Цикл стрельбы: заряжаем патрон



Цикл стрельбы: ждем команды



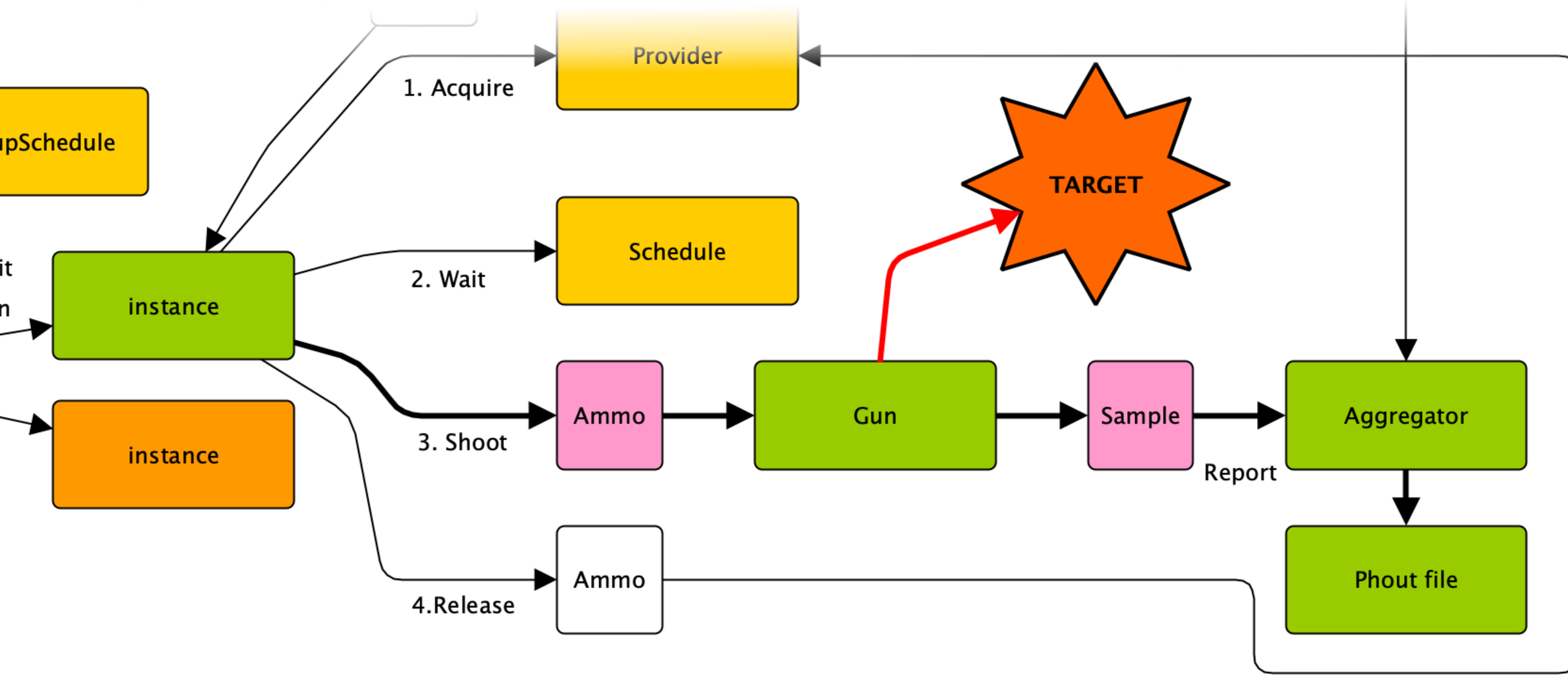
Расписание стрельбы

- работает как ограничение максимального рейта
- помним про закон Литтла, если хотим выдерживать расписание

```
pools:
  - ...
    rps:                              # расписание стрельбы
      type: line                      # линейный рост
      from: 1                         # от 1 RPS
      to: 100000                      # до 100000 RPS
      duration: 60s                   # за 60 секунд

    startup:                          # расписание старта инстансов
      type: once                      # запустить 40000 инстансов за раз
      times: 10000
```

Цикл стрельбы: огонь!

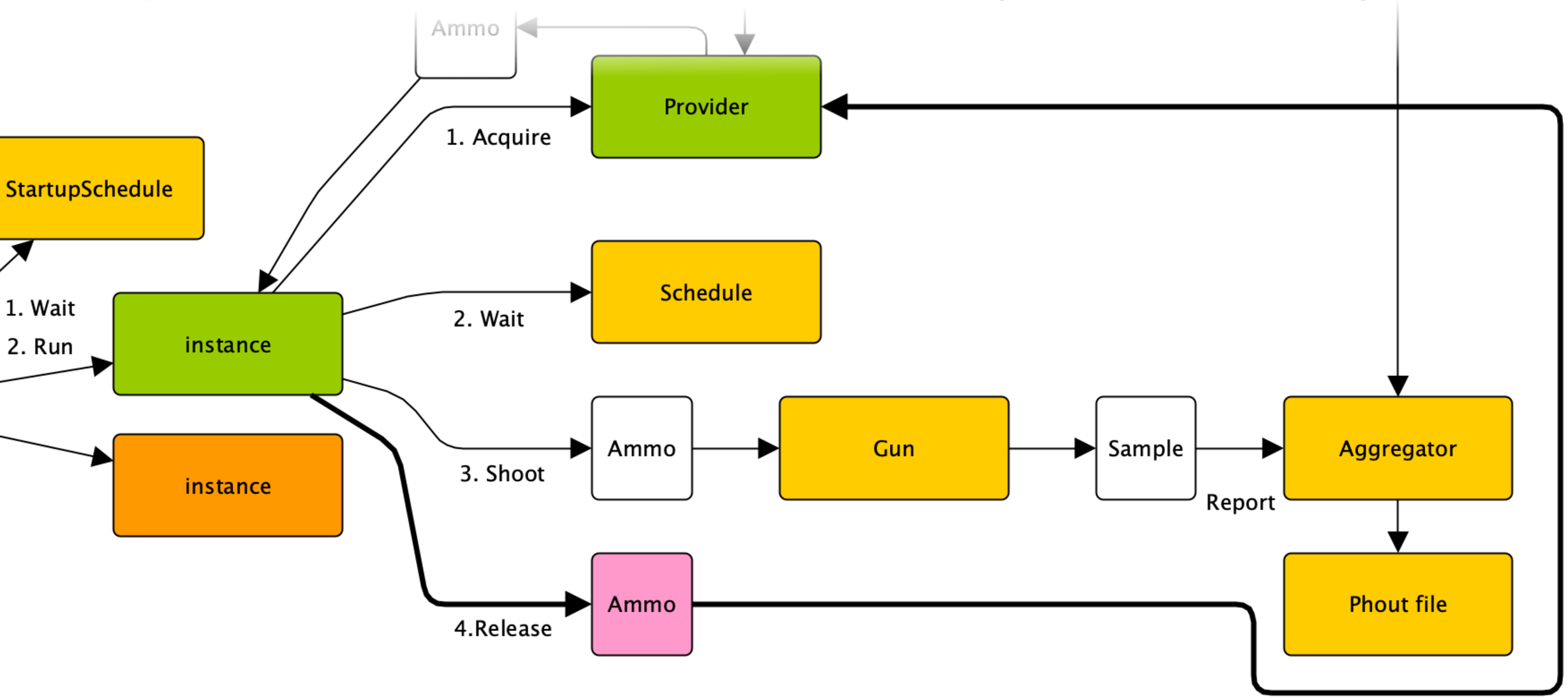


Пушка

› пушка — это место, где описана бизнес-логика “выстрела” и протокол, по которому ведется стрельба

```
pools:
  - id: HTTP pool                    # название пула (пулов может быть несколько)
    gun:
      type: http                     # тип пушки
      target: example.org:80         # куда стрелять
    ammo:
      type: uri                      # формат патронов
      file: ./ammo.uri               # файл с патронами
    result:
      type: phout                    # формат отчета
      destination: ./phout.log       # файл отчета
    rps:                             # расписание стрельбы
      type: line                     # линейный рост
      from: 1                        # от 1 RPS
      to: 100000                     # до 100000 RPS
```

Цикл стрельбы: отдаем “гильзу” на зарядку

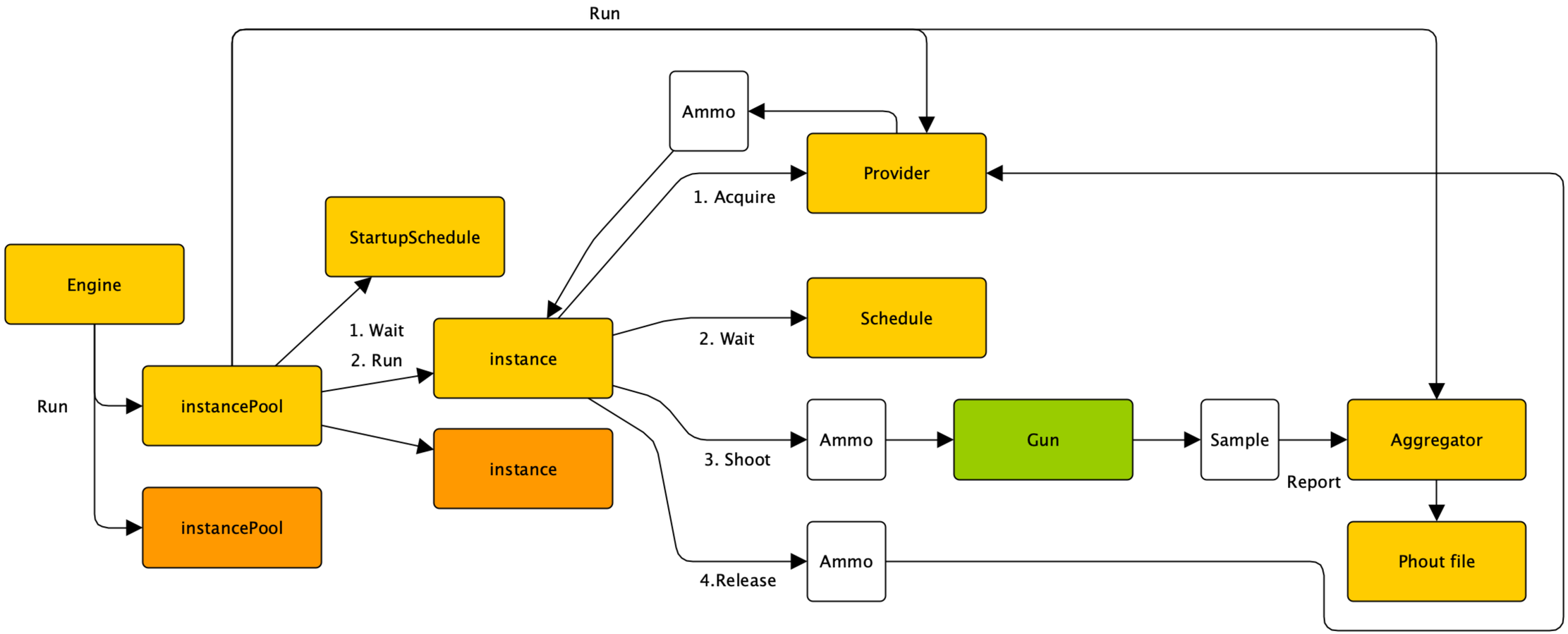


sync.Pool

> чтобы не нагружать Garbage Collector лишними задачами, Pandora старается выделять меньше памяти и переиспользовать уже выделенную

Пишем свою пушку

Что нужно для своей пушки



Что нужно для своей пушки

- описать бизнес-логику выстрела (Gun)
- импортировать остальные кубики из Pandora

Tutorial: github.com/yandex/pandora/blob/develop/docs/custom.rst

Компоненты пушки

```
import ( /* imports */ )

type Ammo struct { /* ammo with custom fields */ }
type GunConfig struct { /* gun config with custom fields */ }
type Gun struct { /* gun with custom components */ }

func (g *Gun) Bind(aggr core.Aggregator, deps core.GunDeps) error {
        // Gun initialization (connect to server)
}

func (g *Gun) case1_method(client pb.MyClient, ammo *Ammo) int { }
func (g *Gun) case2_method(client pb.MyClient, ammo *Ammo) int { }

func (g *Gun) Shoot(ammo core.Ammo) { /* convert ammo type */ }
func (g *Gun) shoot(ammo *Ammo) { /* choose scenario and shoot */ }

func main() { /* get all pieces together */ }
```

Импорты

```
import ( /* imports */ )

type Ammo struct { /* ammo with custom fields */ }
type GunConfig struct { /* gun config with custom fields */ }
type Gun struct { /* gun with custom components */ }

func (g *Gun) Bind(aggr core.Aggregator, deps core.GunDeps) error {
        // Gun initialization (connect to server)
}

func (g *Gun) case1_method(client pb.MyClient, ammo *Ammo) int { }
func (g *Gun) case2_method(client pb.MyClient, ammo *Ammo) int { }

func (g *Gun) Shoot(ammo core.Ammo) { /* convert ammo type */ }
func (g *Gun) shoot(ammo *Ammo) { /* choose scenario and shoot */ }

func main() { /* get all pieces together */ }
```

Импорты

```
import (
        // pandora imports
        ...
        // other lib imports
        ...
        // generated gRPC lib import:
        pb “./my_contracts"
)
```

Где взять контракты gRPC

.proto файл — это описание вашего сервиса: его методов и типов данных

- написать или найти .proto файл от вашего сервиса
- запустить утилиту **protoc**, которая сгенерит код
- код подложить в папку **my_contracts** с соответствующим названием пакета

Tutorial: grpc.io/docs/tutorials/basic/go/

Объявляем компоненты

```
import ( /* imports */ )

type Ammo struct { /* ammo with custom fields */ }
type GunConfig struct { /* gun config with custom fields */ }
type Gun struct { /* gun with custom components */ }

func (g *Gun) Bind(aggr core.Aggregator, deps core.GunDeps) error {
        // Gun initialization (connect to server)
}

func (g *Gun) case1_method(client pb.MyClient, ammo *Ammo) int { }
func (g *Gun) case2_method(client pb.MyClient, ammo *Ammo) int { }

func (g *Gun) Shoot(ammo core.Ammo) { /* convert ammo type */ }
func (g *Gun) shoot(ammo *Ammo) { /* choose scenario and shoot */ }

func main() { /* get all pieces together */ }
```

Патроны

```
type Ammo struct {
        Tag         string
        Param1      string
        Param2      string
        Param3      string
}
```

Файл с патронами json.ammo:

```
...
{"tag": "/MyCase1", "Param1": "146837693,146837692,146837691"}
...
```

Описание полей конфига пушки

```
type GunConfig struct {
        Target string `validate:"required"`
}
```

Пушка: дефолтные поля + наши

```
type Gun struct {
        client grpc.ClientConn // we'll need a gRPC client
        conf   GunConfig
        ...
}
```

Инициализация пушки

```
import ( /* imports */ )

type Ammo struct { /* ammo with custom fields */ }
type GunConfig struct { /* gun config with custom fields */ }
type Gun struct { /* gun with custom components */ }

func (g *Gun) Bind(aggr core.Aggregator, deps core.GunDeps) error {
        // Gun initialization (connect to server)
}

func (g *Gun) case1_method(client pb.MyClient, ammo *Ammo) int { }
func (g *Gun) case2_method(client pb.MyClient, ammo *Ammo) int { }

func (g *Gun) Shoot(ammo core.Ammo) { /* convert ammo type */ }
func (g *Gun) shoot(ammo *Ammo) { /* choose scenario and shoot */ }

func main() { /* get all pieces together */ }
```

Соединение с сервером при инициализации

```
func (g *Gun) Bind(aggr core.Aggregator, deps core.GunDeps) error {
        conn, err := grpc.Dial(
                g.conf.Target,
                grpc.WithInsecure(),
                grpc.WithTimeout(time.Second),
                grpc.WithUserAgent("load test, pandora custom shooter"))
        ...
        g.client = *conn
        ...
}
```

Наши сценарии

```
import ( /* imports */ )

type Ammo struct { /* ammo with custom fields */ }
type GunConfig struct { /* gun config with custom fields */ }
type Gun struct { /* gun with custom components */ }

func (g *Gun) Bind(aggr core.Aggregator, deps core.GunDeps) error {
        // Gun initialization (connect to server)
}

func (g *Gun) case1_method(client pb.MyClient, ammo *Ammo) int { }
func (g *Gun) case2_method(client pb.MyClient, ammo *Ammo) int { }

func (g *Gun) Shoot(ammo core.Ammo) { /* convert ammo type */ }
func (g *Gun) shoot(ammo *Ammo) { /* choose scenario and shoot */ }

func main() { /* get all pieces together */ }
```

Что-то спросили у сервера в сценарии

```
func (g *Gun) case1_method(client pb.MyClient, ammo *Ammo) int {
        ...
        out, err := client.GetSomeData(
                context.TODO(), &pb.ItemsRequest{
                        itemIDs})
        ...
}
```

Метод shoot: выбор сценария и его запуск

```
import ( /* imports */ )

type Ammo struct { /* ammo with custom fields */ }
type GunConfig struct { /* gun config with custom fields */ }
type Gun struct { /* gun with custom components */ }

func (g *Gun) Bind(aggr core.Aggregator, deps core.GunDeps) error {
        // Gun initialization (connect to server)
}

func (g *Gun) case1_method(client pb.MyClient, ammo *Ammo) int { }
func (g *Gun) case2_method(client pb.MyClient, ammo *Ammo) int { }

func (g *Gun) Shoot(ammo core.Ammo) { /* convert ammo type */ }
func (g *Gun) shoot(ammo *Ammo) { /* choose scenario and shoot */ }

func main() { /* get all pieces together */ }
```

Метод shoot: выбор сценария и его запуск

```
sample := netsample.Acquire(ammo.Tag) /* start timer */
...
switch ammo.Tag {
case "/MyCase1":
    code = g.case1_method(client, ammo)
case "/MyCase2":
    code = g.case2_method(client, ammo)
default:
    code = 404
}

sample.SetProtoCode(code) /* stop timer and set response code */
g.aggr.Report(sample) /* report to aggregator */
```

Точка входа

```
import ( /* imports */ )

type Ammo struct { /* ammo with custom fields */ }
type GunConfig struct { /* gun config with custom fields */ }
type Gun struct { /* gun with custom components */ }

func (g *Gun) Bind(aggr core.Aggregator, deps core.GunDeps) error {
        // Gun initialization (connect to server)
}

func (g *Gun) case1_method(client pb.MyClient, ammo *Ammo) int { }
func (g *Gun) case2_method(client pb.MyClient, ammo *Ammo) int { }

func (g *Gun) Shoot(ammo core.Ammo) { /* convert ammo type */ }
func (g *Gun) shoot(ammo *Ammo) { /* choose scenario and shoot */ }

func main() { /* get all pieces together */ }
```

main: регистрируем источник патронов и пушку

```
func main() {
        ...
        coreimport.RegisterCustomJSONProvider(
            "custom_provider", func() core.Ammo { return &Ammo{} })

        register.Gun("My_custom_gun_name", NewGun, func() GunConfig {
                return GunConfig{
                        Target: "default target",
                }
        })
        ...
}
```

Собираем пушку

```
~/tmp » go build
main.go:9:9: cannot find package "github.com/satori/go.uuid" in any of:
    /usr/local/Cellar/go/1.12.3/libexec/src/github.com/satori/go.uuid (from $GOROOT)
    /Users/direvius/go/src/github.com/satori/go.uuid (from $GOPATH)
```

Если не хватает библиотек, go get

```
~/tmp » go build
main.go:9:9: cannot find package "github.com/satori/go.uuid" in any of:
    /usr/local/Cellar/go/1.12.3/libexec/src/github.com/satori/go.uuid (from $GOROOT)
    /Users/direvius/go/src/github.com/satori/go.uuid (from $GOPATH)

~/tmp » go get github.com/satori/go.uuid
```

Исправляем конфиг, запускаем наш бинарь

```yaml
pools:
    - id: HTTP pool
      gun:
          type: My_custom_gun_name
          target: "your_grpc_host:your_grpc_port"
      ammo:
          type: custom_provider
          source:
              type: file
              path: ./json.ammo
      ...
```

Выводы

- Pandora — это еще один сценарный генератор нагрузки
- можно ожидать производительности в несколько десятков тысяч RPS с одной машины с уровнем параллелизма в сотни тысяч виртуальных пользователей
- сценарии нужно будет писать на Go, придется их отлаживать и оптимизировать
- Pandora можно подключить к Яндекс.Танку, тогда получите автоматизацию и мониторинги
- результаты можно заливать в Overload, тогда получите красивые отчеты с графиками

Осторожно!

- разложены грабли. Не такой зрелый проект, как фантом. Относитесь к результатам с долей скептицизма
- можно разложить грабли самому себе. Ваши тесты — код
- сценарии нужно писать с оглядкой на производительность. Возможно даже профилировать
- следите за GC (хотя по опыту это всплывает нечасто)

Яндекс

Спасибо

Алексей Лавренюк

direvius@yandex-team.ru

Ссылки

- Pandora: github.com/yandex/pandora
- Pandora docs: yandexpandora.readthedocs.io/en/develop
- Go scheduler: rakyll.org/scheduler/
- High performance Go workshop: dave.cheney.net/high-performance-go-workshop/dotgo-paris.html
- Go chanels: codeburst.io/diving-deep-into-the-golang-channels-549fd4ed21a8
- Go memory allocator: blog.learngoprogramming.com/a-visual-guide-to-golang-memory-allocator-from-ground-up-e132258453ed