Обозначения

ПО — ...

API — ...

БД — ...

3 Разработка технологии тестирования

3.1 Выбор подходов и методов тестирования

Процесс разработки программного обеспечения в том виде, как оно определяется в современной модели жизненного цикла программного обеспечения предполагает три стадии тестирования [1]:

- автономное тестирование компонентов ПО;
- комплексное тестирование разрабатываемого ПО;
- системное или оценочное тестирование на соответствие основным критериям качества.

Для проведения автономного и комплексного тестирования необходимо сформировать тестовые наборы, опираясь на структурный или функциональный подход.

Структурный подход базируется на том, что известна структура тестируемого ПО, в том числе его алгоритмы. Тесты строят так, чтобы обеспечить максимальное покрытие исходного кода.

Функциональный подход основывается на том, что структура ПО не известна. В этом случае тесты строят, опираясь на функциональные спецификации. Тесты строят на базе различных способов декомпозиции множества данных.

Разработанное ПО включает в себя разнородные алгоритмы, для всестороннего тестирования которых с помощью структурного подхода понадобились бы значительные затраты времени на изучение исходного кода, разработку большого числа тестов и заглушек. По этой причине было решено использовать функциональный подход, который позволил бы значительно сократить время на разработку тестов, обеспечивая при это тестирование всей необходимой функциональности [2].

В качестве оценочного тестирования согласно ТЗ было выбрано нагрузочное тестирование.

3.2 Разработка плана автономного тестирования

3.3 Разработка плана комплексного тестирования

3.4 Выбор языка и библиотек для функционального тестирования

Так как написание тестов на Golang требует значительного времени и такие тесты сложнее поддерживать в силу непопулярности языка среди тестировщиков, было решено тестировать разработанные микросервисы, предварительно запустив их (см. приложение X) и обращаясь к ним по протоколу HTTP. Такой подход позволил реализовать тесты не привязываясь к языку реализации исходного ПО.

Поскольку Python обладает простым синтаксисом, большим количеством библиотек и популярен среди тестировщиков (рисунок X), именно он был выбран для реализации тестов [3].

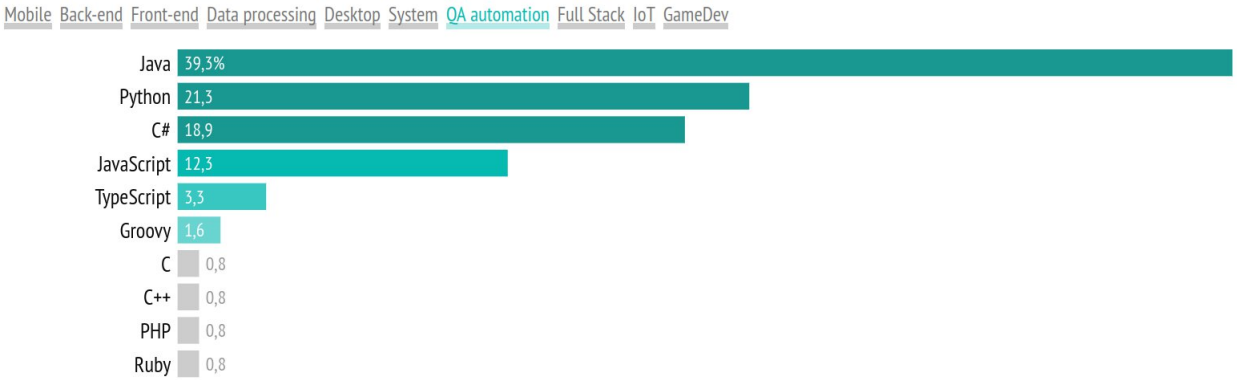


Рисунок X — наиболее популярные языки в области автоматизированного тестирования

В качестве основной библиотеки для тестирования была выбрана библиотека pytest, являющаяся одной из наиболее популярных библиотек для автоматизированного тестирования [4].

Pytest обладает следующими основными преимуществами [5]:

- меньше повторяющегося кода за счет независимости от API;
- выполнение определенного набора тестов с помощью фильтрации;
- параметризация тестов — запуск одного и того же теста с разными наборами параметров;
- гибкость — архитектура библиотеки основана на плагинах, которые можно установить отдельно;

– полная обратная совместимость с unittest — возможность запуска тестов, написанных на нем;
– выполнение нескольких тестов параллельно;
– установочный код можно использовать повторно.

В дополнение к pytest была использована библиотека allure, формирующая интерактивные отчеты о прохождении тестов. Тесты в allure можно иерархически группировать и сопровождать логами и вложениями. Allure поддерживается не только для Python, но и для Java, JavaScript, Ruby, PHP , .Net и Scala.

Такой широкий набор поддерживаемых языков программирования делает allure (рисунок X) знакомым многим разработчикам, тестировщикам и менеджерам, что упрощает поддержку тестов [6].

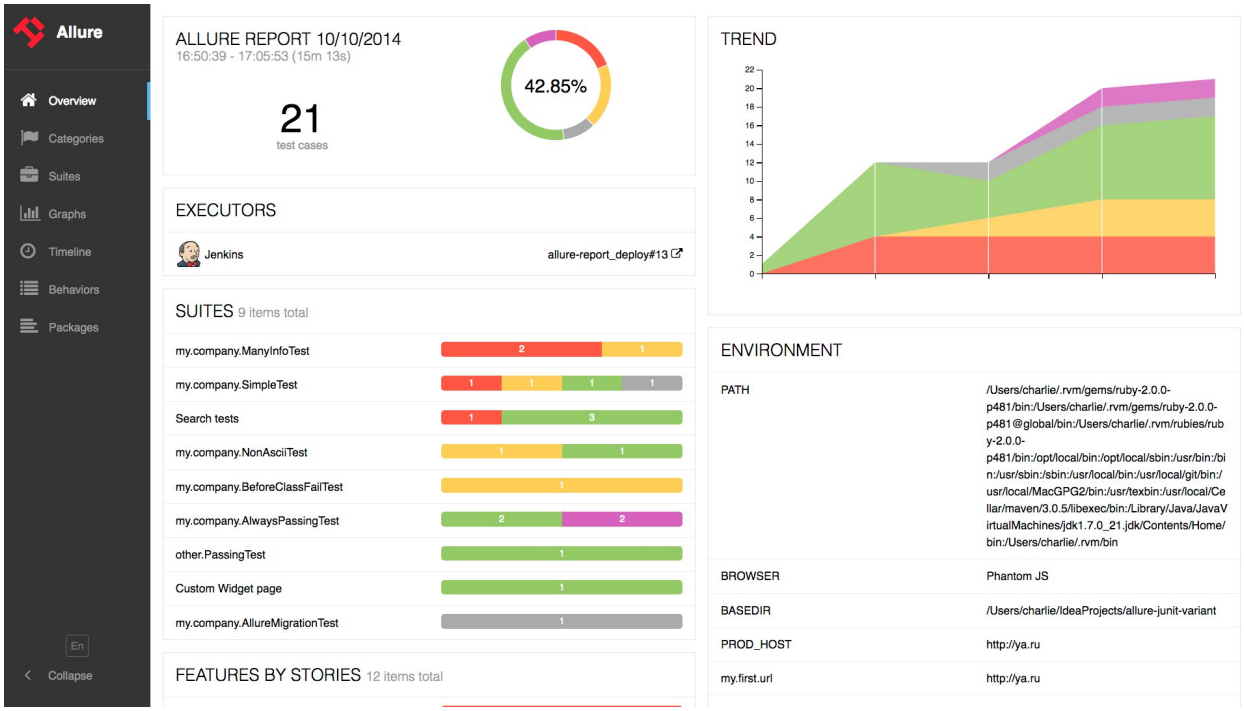


Рисунок X — интерфейс allure

3.5 Реализация и проведение функциональных тестов

Для упрощения написания тестов и генерации отчетов был реализован вспомогательный модуль utils.py, отвечающий за отправку http-запросов к микросервисам, проверку http-ответов и их прикрепление к отчетам в allure. Программный код utils.py приведен в листинге X.

Листинг X — программный код utils.py

```
import requests
import allure
import json

@allure.step("Send request") # отправка запроса - новый этап  в отчете
def send_request(port, url, payload):
    path = f"http://127.0.0.1:{port}/{url}"
    allure.attach(path, 'Request URL', allure.attachment_type.TEXT)
    # прикрепить данные запроса к отчету
    allure.attach(json.dumps(payload, indent=4, ensure_ascii=False).encode(), 'Request
payload', allure.attachment_type.TEXT)
    resp = requests.post(path, json = payload)
    # прикрепить данные ответа к отчету
    allure.attach(json.dumps(resp.json(), indent=4, ensure_ascii=False).encode(), 'Response
payload', allure.attachment_type.TEXT)
    return resp

@allure.step("Check response [ok]") # проверка ответа без ошибок
def is_ok_response(response):
    allure.attach(json.dumps(response.json(), indent=4, ensure_ascii=False).encode(),
'Response payload', allure.attachment_type.TEXT)
    return response.status_code == 200 \
        and response.json()["status_code"] == 200 \
            and response.json()["status_str"] == "ok"

@allure.step("Check response [error]") # проверка ответа при ошибке
def is_error_response(response):
    allure.attach(json.dumps(response.json(), indent=4, ensure_ascii=False).encode(),
'Response payload', allure.attachment_type.TEXT)
    return response.status_code != 200 \
        and response.json()["status_code"] != 200 \
            and response.json()["status_str"] == "error"

def ordered_json(obj): # упорядочивание JSON для последующего сравнения
    if isinstance(obj, dict):
        return sorted((k, ordered_json(v)) for k, v in obj.items())
    if isinstance(obj, list):
        return sorted(ordered_json(x) for x in obj)
    else:
        return obj
```

Помимо utils.py, были реализованы вспомогательные модули settings.py и consts.py, которые определяют порты по которым доступны микросервисы и данные для HTTP-запросов в тестах.

Пример реализации простейшего теста (проверка работы анализатора для неправильно решенного пользователем теста с одним вариантом ответа) приведен в листинге X.

Листинг X — пример реализации теста

```
import utils.settings as settings
import utils.utils as utils
from utils.consts import *

import allure
import copy

# < ...>

# определение иерархии и описания в отчете
@allure.description("Test for singlechoice wrong-answered task")
@allure.epic("Unit-testing")
@allure.story("Analyzer")
def test_single_correct_negative():
    # считывание данных при правильном ответе из констант
    payload = copy.deepcopy(Analyzer.single_valid_positive)
    # изменение ID выбранного ответа
    payload["data"]["user_answer_id"] = 1
    # http-запрос
    resp = utils.send_request(settings.ANALYZER_PORT,
                              "check", payload)
    # http-ответ успешен?
    assert utils.is_ok_response(resp)
    # http-ответ задание решено неверно?
    assert resp.json()["is_correct"] == False
    # возвращена ожидаемая подсказка?
    assert resp.json()[
        "data"]["hint"] == payload["data"]["task"]["answers"][1]["hint"]

# < ...>
```

Пример отчета, полученного после выполнения этого теста (и аналогичных ему) приведен на рисунках X-X.

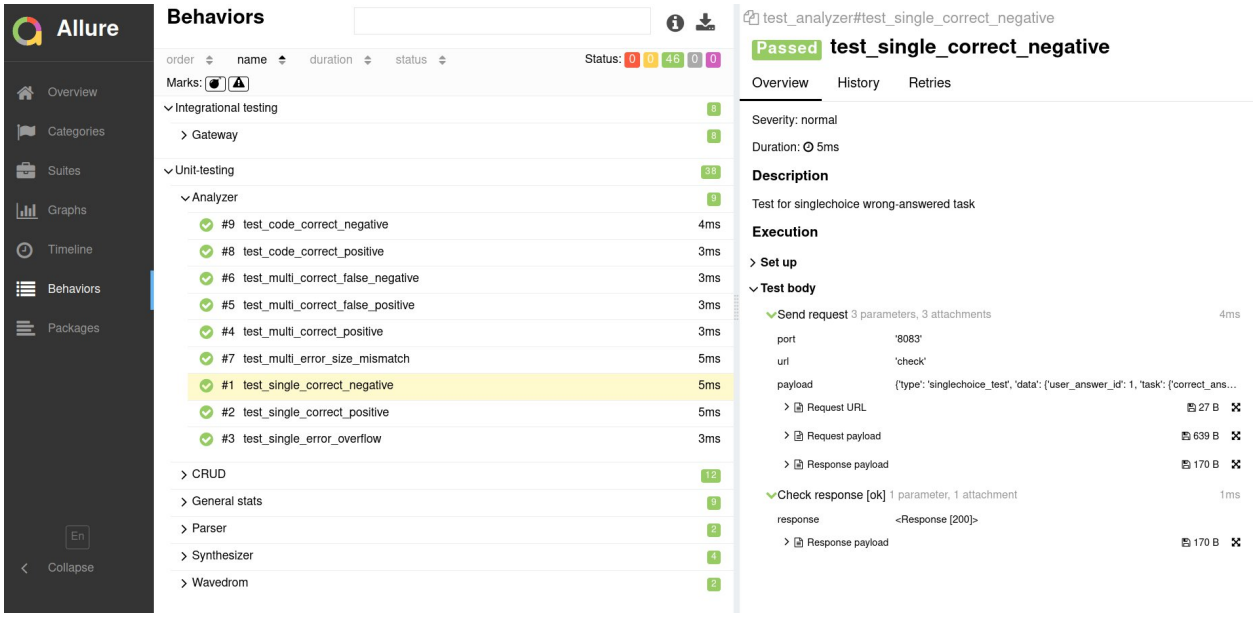


Рисунок X — отчет о результате теста

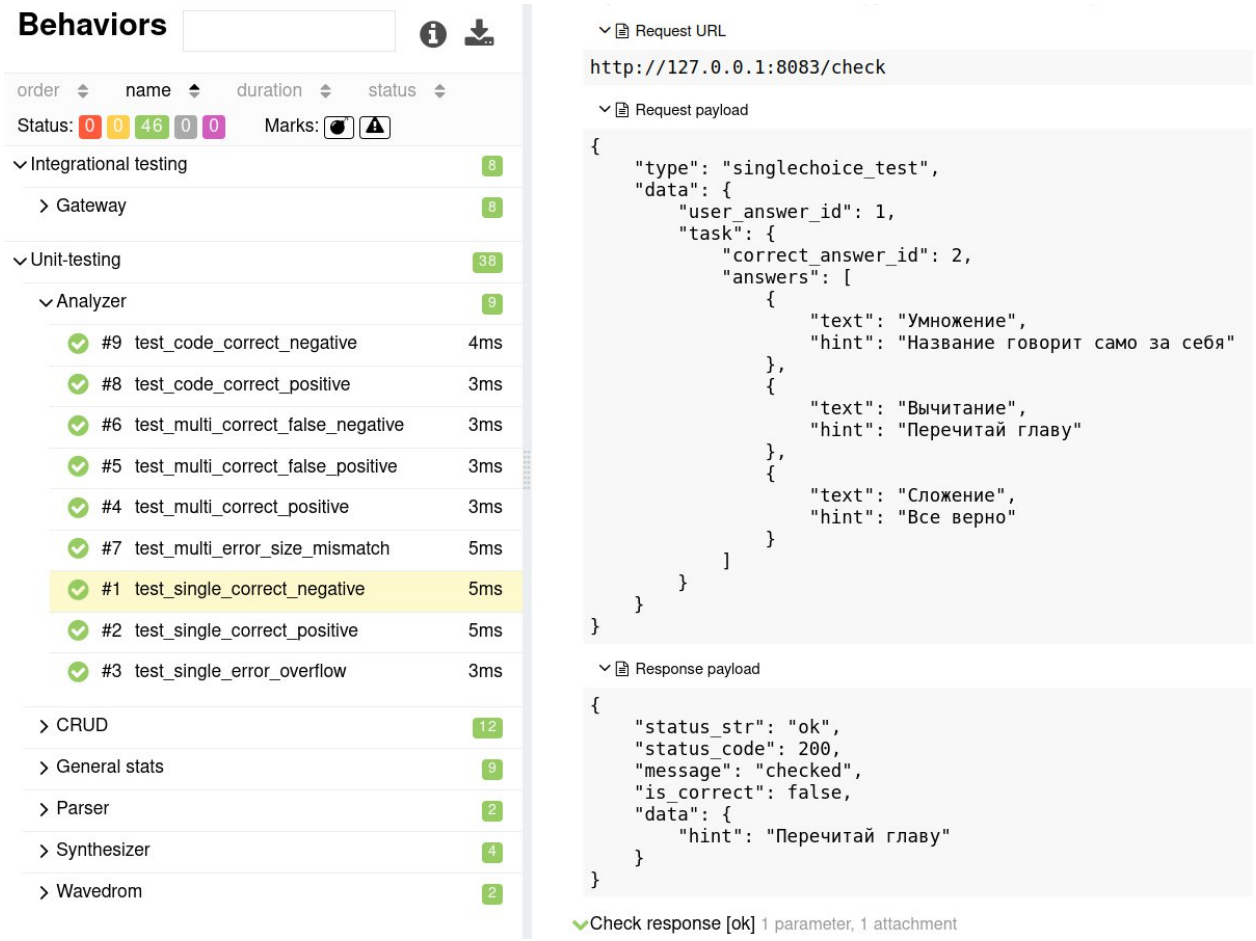


Рисунок X — приложения к отчету

После реализации аналогичным образом всех тестов из таблиц M-N они были запущены.

Благодаря параллельному запуску тестов через плагин xdist (в данном случае — в 2 потока) удалось выполнить все тесты менее, чем за 600 мс (рисунок X).

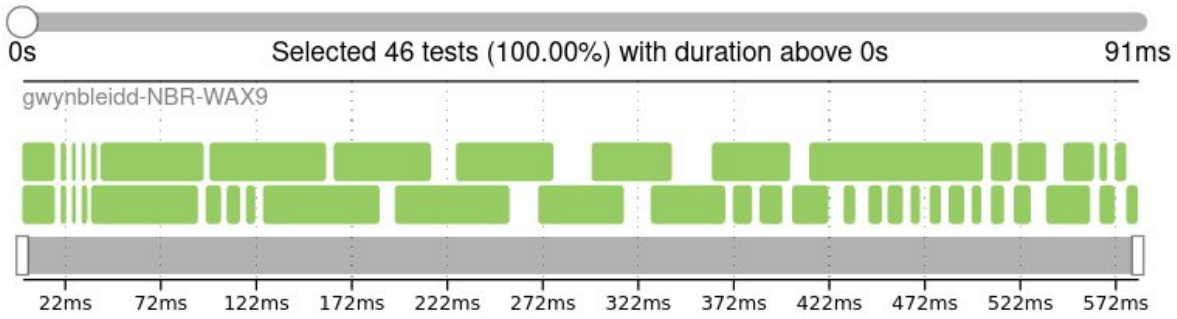


Рисунок X — хронология запуска тестов

Статистика запуска тестов из отчета allure приведена на рисунке X.

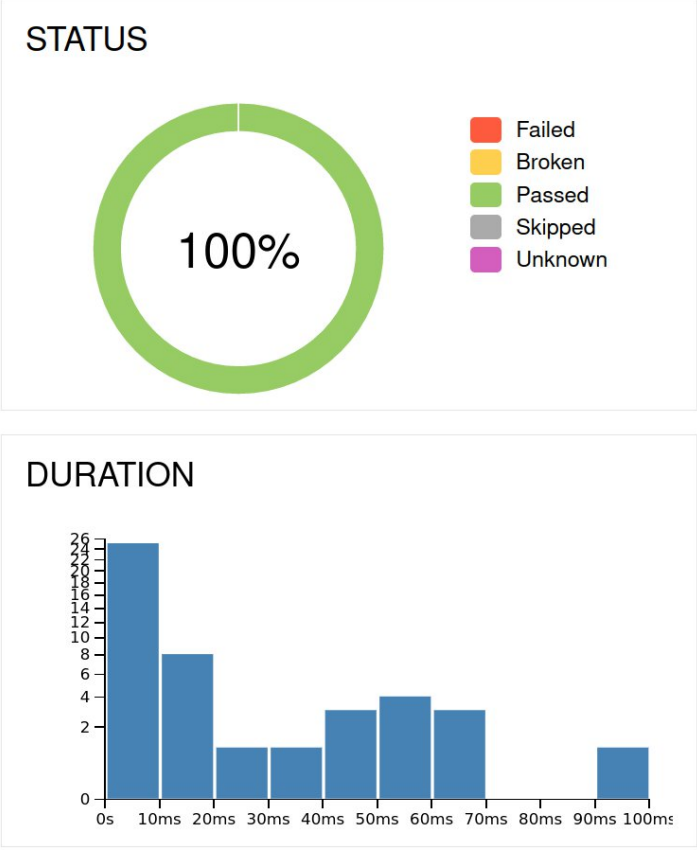


Рисунок X — статистика запуска тестов

Все тесты завершились успешно, о чем свидетельствует приведенная статистика.

3.5 Нагрузочное тестирование

Для упрощения поддержки тестов реализовать нагрузочное тестирование было решено так же с помощью Python. Наиболее популярной библиотекой для нагрузочного тестирования с помощью Python является locust. Locust предоставляет веб-интерфейс для настройки нагрузки и просмотра отчетов.

Пользователь значительно чаще считывает данные из БД, чем пишет в нее, кроме того значительное время может занимать синтез устройств и преобразование временных диаграмм при проверке заданий на программирование.

По этим причинам в ходе нагрузочного тестирования были использованы запросы на чтение к БД, отправленные через компонент бизнес-логики (чтобы учесть задержку при проксировании запроса), а так же запросы на синтез устройств и преобразование диаграмм непосредственно к соответствующим микросервисам.

Конфигурация оборудования, на котором проводится нагрузочное тестирование:

- ОС: Ubuntu 20.04 focal
- Ядро: x86_64 Linux 5.15.0-60-generic
- Процессор: Intel Core i3-10110U @ 4x 4,1 ГГц
- ОЗУ: 4229 МБ / 7691 МБ

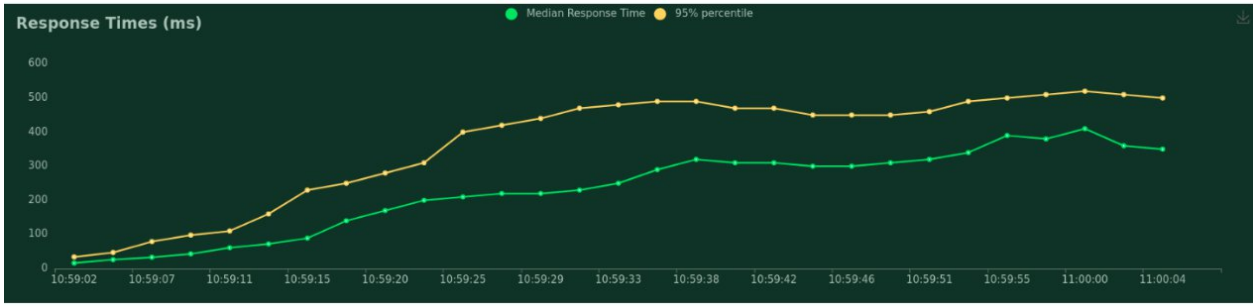
Так как конфигурация оборудования уступает требуемой в ТЗ, для нагрузочного тестирования была взята максимальная нагрузка в 100 пользователей.

Результаты нагрузочного тестирования преобразователя временных диаграмм приведены в таблице X.

Таблица X — результаты нагрузочного тестирования преобразователя временных диаграмм

Статистика запросов							
Запросы	Ошибки	Среднее (мс)	Мин. (мс)	Макс. (мс)	Сред. размер (байт)	RPS	Ошибки / с
18458	0	261	3	706	493	279.9	0.0
Статистика ответов							
50%ile (мс)	60%ile (мс)	70%ile (мс)	80%ile (мс)	90%ile (мс)	95%ile (мс)	99%ile (мс)	100%ile (мс)
260	300	350	400	450	470	520	710

Изменения частоты запросов, времени ответа и количества пользователей показаны на рисунке X.



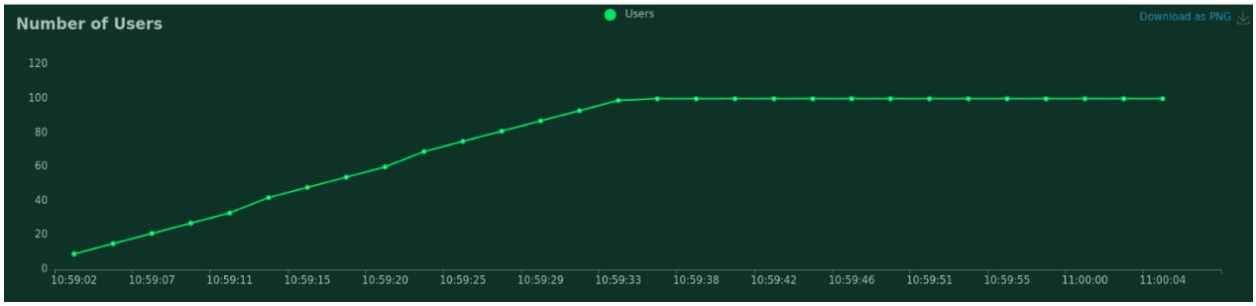


Рисунок X — результаты нагрузочного тестирования преобразователя временных диаграмм

Результаты нагрузочного тестирования синтезатора приведены в таблице X.

Таблица X — результаты нагрузочного тестирования синтезатора

Статистика запросов							
Запросы	Ошибки	Среднее (мс)	Мин. (мс)	Макс. (мс)	Сред. размер (байт)	RPS	Ошибки / с
12936	0	525	13	1339	3050	152.8	0.0
Статистика ответов							
50%ile (мс)	60%ile (мс)	70%ile (мс)	80%ile (мс)	90%ile (мс)	95%ile (мс)	99%ile (мс)	100%ile (мс)
580	610	640	680	770	870	950	1300

Изменения частоты запросов, времени ответа и количества пользователей показаны на рисунке X.

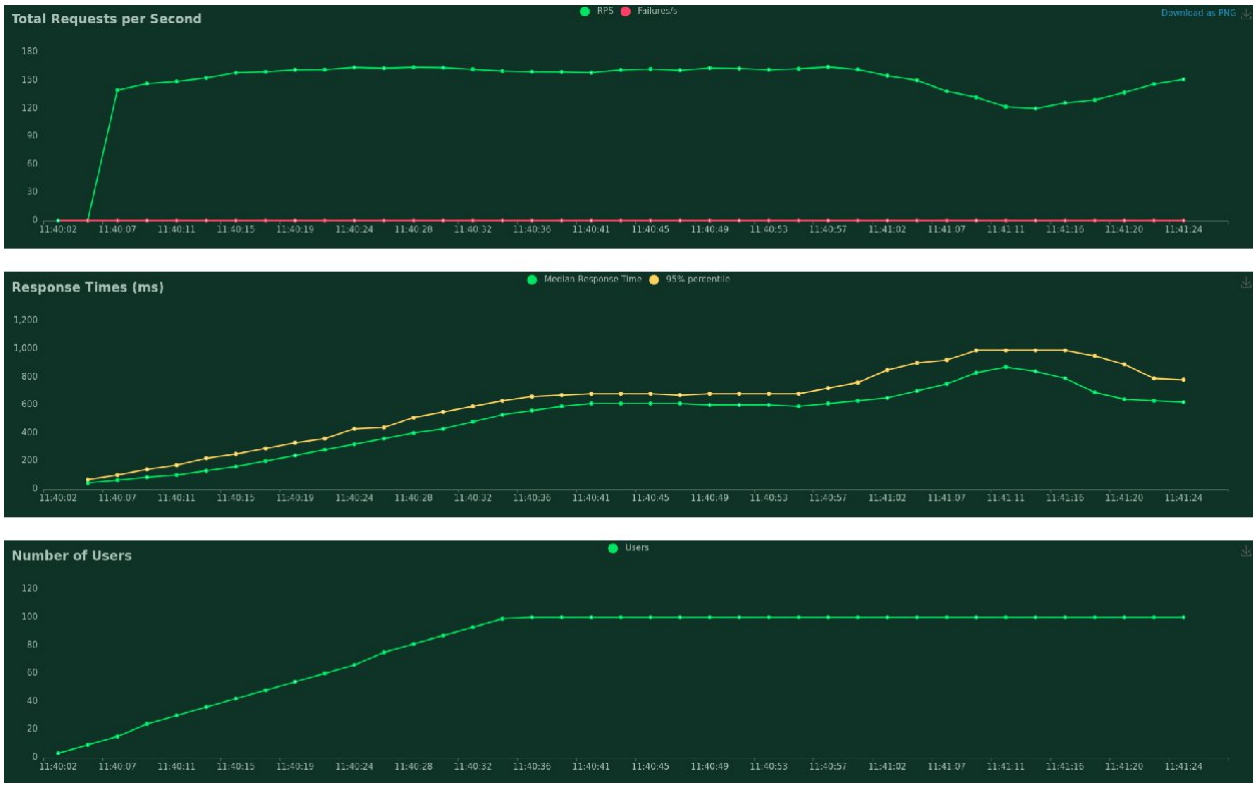


Рисунок X — результаты нагрузочного тестирования синтезатора

Результаты нагрузочного тестирования обращения к БД через слой бизнес-логики приведены в таблице X.

Таблица X — результаты нагрузочного тестирования обращения к БД через слой бизнес-логики

Статистика запросов								
Маршрут	Запросы	Ошибки	Среднее (мс)	Мин. (мс)	Макс. (мс)	Сред. размер (байт)	RPS	Ошибки / с
/levels	6743	0	85	3	676	506	111.8	0.0
/stats	20486	0	161	2	1151	331	339.5	0.0
Итого	27229	0	142	2	1151	374	451.3	0.0
Статистика ответов								
Маршрут	50%ile (мс)	60%ile (мс)	70%ile (мс)	80%ile (мс)	90%ile (мс)	95%ile (мс)	99%ile (мс)	100%ile (мс)
/levels	78	94	110	130	160	180	220	680
/stats	110	140	190	270	370	460	630	1200
Итого	100	130	160	210	330	430	600	1200

Изменения частоты запросов, времени ответа и количества пользователей показаны на рисунках X-X.

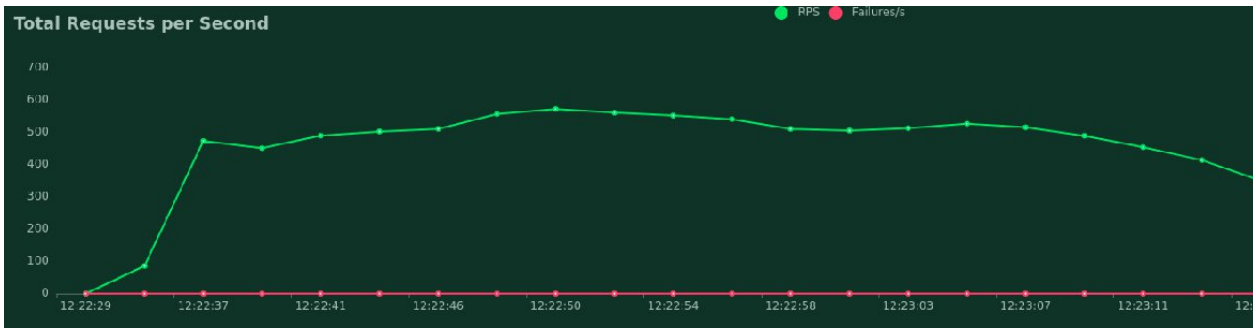


Рисунок X — результаты нагрузочного тестирования обращения к БД через слой бизнес-логики

Рисунок X — результаты нагрузочного тестирования обращения к БД через слой бизнес-логики

Источники

1. Книга Ивановой
2. https://coderlessons.com/tutorials/kachestvo-programmnogo-obespecheniia/ruchnoe-testirovanie/chernyi-iashchik-protiv-belaia-korobka
3. https://habr.com/ru/post/543346/
4. https://www.softwaretestinghelp.com/python-testing-frameworks/
5. https://blog.skillfactory.ru/glossary/pytest/
6. https://docs.qameta.io/allure/
7.