Министерство образования и науки Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н.Г. ЧЕРНЫШЕВСКОГО»

Кафедра дискретной математики и информационных технологий

**Разработка обучающей платформы для решения задач на языке программирования С++**

НАУЧНО-ИССЛЕДОВАТЕЛЬСКАЯ РАБОТА

студентов 2 курса 221 группы

специальности 09.03.01 «Информатика и вычислительная техника»

факультета компьютерных наук и информационных технологий

Мусатова Федора Алексеевича, Васильевой Софии Алексеевны, Беляева Владислава Алексеевич, Блохина Артёма Романовича

|  |  |  |
| --- | --- | --- |
| Научный руководитель  старший преподаватель | \_\_\_\_\_\_\_\_\_\_ | А.А. Трунов |
|  | подпись, дата |  |
| Заведующий кафедрой  доцент, к.ф.-м.н. | \_\_\_\_\_\_\_\_\_\_ | Л.Б. Тяпаев |
|  | подпись, дата |  |

Саратов 2025

**СОДЕРЖАНИЕ**

**Оглавление**

[ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ 3](#_Toc198382303)

[ВВЕДЕНИЕ 4](#_Toc198382304)

[1. Краткий обзор ключевых технологий учебной платформы для решения задач на языке C++ 6](#_Toc198382305)

[**1.1. Микросервисная архитектура и язык программирования Go** 6](#_Toc198382306)

[**1.2. Объектное хранилище MinIO** 6](#_Toc198382307)

[**1.3. Dragonfly как брокер сообщений** 7](#_Toc198382308)

[**1.4. Клиентская часть на React** 7](#_Toc198382309)

[**1.5. Взаимодействие компонентов и преимущества микросервисного стека** 8](#_Toc198382310)

[2. Анализ требований, проектирование прикладного решения и реализация микросервисного решения 9](#_Toc198382311)

[**2.1. Анализ требований** 9](#_Toc198382312)

[**2.2. Проектирование архитектуры прикладного решения** 10](#_Toc198382313)

[**2.3 Реализация интеграции и проектирование микросервисного решения GoRunner** 10](#_Toc198382314)

[**2.3.1 Монорепозиторий и общее устройство проекта** 11](#_Toc198382315)

[**2.3.2 Детальная конфигурация MinIO** 11](#_Toc198382316)

[**2.3.3 Сервис работы с MinIO: полный пример** 12](#_Toc198382317)

[**2.3.4 Формат и размещение тестовых наборов (Listing 1)** 15](#_Toc198382318)

[**2.3.5 Публикация задач: структура и пример (Listing 2)** 15](#_Toc198382319)

[**2.3.6 Микросервис Coderunner: полная логика (Listing 3)** 16](#_Toc198382320)

[**2.3.7 Контейнеризация и развёртывание** 18](#_Toc198382321)

[**ЗАКЛЮЧЕНИЕ** 20](#_Toc198382322)

[СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ 23](#_Toc198382323)

# **ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ**

Amazon S3 — облачное объектное хранилище компании Amazon Web Services для хранения и управления данными[1].

REST API — программный интерфейс, использующий HTTP-запросы для взаимодействия с сервисами.

Бакеты (buckets) — это основной контейнер для хранения данных в сервисе Amazon S3. Бакеты используются для организации и управления объектами (файлами) в облачном хранилище Amazon Web Services.

AWS Signature Version 4 — это протокол аутентификации, используемый для подписи HTTP-запросов к сервисам Amazon Web Services.

Identity and Access Management — это сервис Amazon Web Services, который позволяет управлять доступом к ресурсам AWS.

MinIO — это распределённая система хранения объектов с поддержкой S3, обеспечивающая масштабируемое и надёжное хранение данных в облаке и на локальных серверах[2].

Жизненный цикл — набор правил для автоматического управление объектами в хранилище объектов.

API — Application Programming Interface.

HTTP — HyperText Transfer Protocol.

IAM — Identity and Access Management.

# **ВВЕДЕНИЕ**

Современные подходы к разработке образовательных платформ требуют не только эффективной методологии обучения, но и применения технологически продвинутой архитектуры, обеспечивающей масштабируемость, отказоустойчивость и высокую производительность. Язык программирования C++ сохраняет ключевое значение в образовательных программах, особенно в контексте подготовки специалистов в сфере алгоритмов, системного и прикладного программирования. В условиях цифровизации образования актуальной становится разработка специализированной учебной платформы, ориентированной на решение задач на C++, с использованием современных технологических решений.

Актуальность проекта обусловлена необходимостью создания интерактивного сервиса, обеспечивающего автоматическую проверку решений, хранение пользовательских данных, гибкое масштабирование нагрузки и высокую доступность. В качестве технологического стека используется язык Go[3] для реализации микросервисной архитектуры, MinIO в роли S3-совместимого хранилища объектов[3], высокопроизводительное in-memory хранилище Dragonfly для кэширования и управления сессиями, а также React для построения современного клиентского интерфейса[4]. Такой выбор технологий обеспечивает гибкость, независимость компонентов и легкость масштабирования.

Новизна решения заключается в комплексной реализации учебной платформы на базе микросервисного подхода, что позволяет эффективно разделять зоны ответственности, ускорять разработку и сопровождение, а также обеспечивать отказоустойчивость. Использование Go обеспечивает высокую производительность серверной части, MinIO — надежное и масштабируемое хранилище пользовательских решений и тестов, а React — интуитивно понятный и отзывчивый пользовательский интерфейс. В совокупности, эти технологии формируют современную и эффективную основу для платформы, способной поддерживать массовое обучение и индивидуальный прогресс каждого пользователя.

Цель работы - Разработка учебной платформы для решения задач на языке программирования C++, основанной на микросервисной архитектуре с использованием современных технологий (Go, MinIO, Dragonfly, React), обеспечивающей автоматическую проверку решений, масштабируемость, отказоустойчивость и удобный пользовательский интерфейс для эффективного освоения практических навыков программирования.

ЗАДАЧИ ИССЛЕДОВАНИЯ

1. Провести обзор существующих учебных платформ и сервисов для решения задач на языке C++, выделив их сильные и слабые стороны.
2. Проанализировать требования к функциональности учебной платформы, включая поддержку автоматической проверки решений, хранение пользовательских данных и масштабируемость.
3. Спроектировать архитектуру системы на основе микросервисного подхода, с использованием языка Go для серверной части, MinIO как объекта хранения решений, Dragonfly для управления сессиями, а также React для клиентского интерфейса.
4. Реализовать основные микросервисы: сервис проверки решений, сервис хранения данных, интерфейс взаимодействия с пользователем.
5. Провести тестирование и отладку платформы, а также разработать рекомендации по её использованию в учебных учреждениях и возможному масштабированию для массового онлайн-доступа.

Разработанная платформа будет полезна для студентов, школьников и преподавателей, позволяя эффективно осваивать язык C++ в интерактивной среде. В дальнейшем проект может быть расширен для поддержки других языков программирования, подключения внешних задачников и реализации гибкой системы прогрессии обучения.

1. **Краткий обзор ключевых технологий учебной платформы для решения задач на языке C++**

**1.1. Микросервисная архитектура и язык программирования Go**

Микросервисная архитектура — современный подход к построению программных систем, при котором приложение разбивается на множество независимых сервисов с узкой специализацией[10]. Каждый микросервис выполняет определённый набор функций и взаимодействует с другими через стандартизированные интерфейсы, чаще всего REST API или протоколы обмена сообщениями.

Использование микросервисного подхода позволяет обеспечить:

* Гибкость и масштабируемость — отдельные сервисы можно разворачивать, обновлять и масштабировать независимо.
* Высокую отказоустойчивость — сбой одного микросервиса не приводит к остановке всей платформы.
* Упрощённое сопровождение и расширение функционала.

Для разработки серверной части выбран язык Go, который имеет следующие преимущества:

1. Высокой производительностью и эффективным управлением ресурсами.
2. Встроенной поддержкой конкурентного программирования через горутины.
3. Простотой синтаксиса и богатой экосистемой для разработки веб-сервисов и микросервисов.

Это позволяет создавать быстрые, масштабируемые и легко поддерживаемые микросервисы для платформы.

**1.2. Объектное хранилище MinIO**

MinIO — это высокопроизводительное распределённое объектное хранилище с открытым исходным кодом, полностью совместимое с API Amazon S3. MinIO обеспечивает:

1. Масштабируемое хранение больших объёмов данных.
2. Высокую доступность и отказоустойчивость.
3. Поддержку современных механизмов безопасности, включая шифрование и контроль доступа.

В рамках учебной платформы MinIO используется для хранения исходных кодов пользователей, тестовых данных, результатов проверок и прочих связанных файлов, что обеспечивает надёжное и масштабируемое хранение данных.

**1.3. Dragonfly как брокер сообщений**

В архитектуре платформы Dragonfly выступает в роли брокера сообщений, обеспечивая надёжный и высокопроизводительный обмен событиями между микросервисами[5]. Это критически важно для обеспечения асинхронного взаимодействия и масштабируемости.

Основные функции Dragonfly в платформе:

* Организация очередей сообщений для передачи заданий на проверку кода, уведомлений о результатах и другой межсервисной коммуникации.
* Гарантированная доставка сообщений и упорядоченность их обработки.
* Снижение связности между сервисами — микросервисы взаимодействуют через обмен сообщениями, что упрощает их независимое развитие и масштабирование.

Использование Dragonfly как брокера сообщений позволяет плавно обрабатывать высокие нагрузки, разгружать синхронные операции и повышать отказоустойчивость платформы.

**1.4. Клиентская часть на React**

React обеспечивает создание интерактивного и отзывчивого пользовательского интерфейса для платформы. Благодаря компонентному подходу и виртуальному DOM, React позволяет:

1. Быстро и эффективно отображать большое количество динамически изменяющихся данных (задачи, результаты проверок, рейтинги).
2. Создавать удобные формы для ввода решений и навигации.
3. Обеспечивать адаптивность интерфейса для различных устройств — от десктопов до мобильных.

Интерфейс на React взаимодействует с серверной частью через REST API[4], получая данные и отправляя решения пользователей на проверку.

**1.5. Взаимодействие компонентов и преимущества микросервисного стека**

Клиент на React отправляет запросы на микросервисы на Go. Микросервисы через Dragonfly обмениваются сообщениями, например, сервис отправляет задачу на проверку кода в очередь, а сервис проверки решений — возвращает результаты. MinIO обеспечивает долговременное хранение всех данных и файлов. Dragonfly гарантирует надёжную доставку и обработку сообщений, позволяя системе эффективно масштабироваться и оставаться отказоустойчивой.

Такой стек технологий позволяет построить гибкую, масштабируемую и устойчивую к нагрузкам учебную платформу, готовую к развитию и внедрению в образовательных организациях.

**2. Анализ требований, проектирование прикладного решения и реализация микросервисного решения**

**2.1. Анализ требований**

При разработке учебной платформы для решения задач на языке C++ были выделены основные функциональные и нефункциональные требования:

Функциональные требования:

1. Возможность регистрации и аутентификации пользователей с разными ролями (студенты, преподаватели, администраторы).
2. Представление учебных заданий и задач с возможностью загрузки исходного кода для решения.
3. Автоматическая проверка решений на корректность и полноту с использованием компилятора C++.
4. Отображение результатов проверки в удобном интерфейсе с детализацией ошибок и предупреждений.
5. Поддержка повторных попыток решения и ведение истории решений пользователя.
6. Управление тестовыми данными и шаблонами проверок.
7. Масштабируемость системы для поддержки большого числа одновременно работающих пользователей.
8. Надёжное хранение всех пользовательских данных и решений с возможностью резервного копирования.

Нефункциональные требования:

1. Высокая производительность и минимальное время отклика при проверке решений.
2. Обеспечение безопасности данных и пользователей, включая защиту от несанкционированного доступа.
3. Возможность горизонтального масштабирования компонентов.
4. Устойчивость к сбоям — платформа должна сохранять работоспособность при отказе отдельных сервисов.
5. Гибкость и расширяемость архитектуры для интеграции новых языков программирования и функциональных модулей.

**2.2. Проектирование архитектуры прикладного решения**

На основе анализа требований была выбрана микросервисная архитектура, обеспечивающая модульность, масштабируемость и независимость компонентов.

Основные компоненты архитектуры:

Сервис управления задачами — хранение и публикация учебных заданий, хранение тестовых данных.

Сервис проверки решений — получение исходного кода, компиляция и запуск тестов, генерация отчётов.

Сервис хранения данных — взаимодействие с объектным хранилищем MinIO для сохранения исходников, тестовых данных и результатов.

Брокер сообщений Dragonfly — обеспечивает асинхронный обмен сообщениями между сервисами, например, очередь заданий на проверку кода и передачу результатов.

Веб-интерфейс на React — взаимодействие с пользователем, отображение задач, загрузка решений, просмотр результатов.

Взаимодействие компонентов:

Пользователь через React-клиент отправляет решение. Сервис проверки решений принимает задачу, отправляет её в очередь Dragonfly. Сервис проверки забирает задачу из очереди, загружает исходный код из MinIO, компилирует и запускает тесты. Затем результаты проверки возвращаются обратно через Dragonfly сервису управления задачами, который сохраняет отчёты в MinIO. После клиент получает обновлённый статус решения через API.

Такое разделение ответственности позволяет оптимизировать нагрузку, легко масштабировать отдельные компоненты и быстро внедрять новые функции.

**2.3 Реализация интеграции и проектирование микросервисного решения GoRunner**

Ниже приводится описание архитектуры, взаимодействия компонентов и примеры кода, используемые в проекте GoRunner. Структура раздела расширена за счёт углублённого разбора каждого шага: от конфигурации внешних сервисов до параллельной обработки задач внутри микросервиса Coderunner.

### **2.3.1 Монорепозиторий и общее устройство проекта**

GoRunner собран в одном репозитории, что позволяет централизованно управлять всеми частями системы и согласованно выпускать новые версии:

gorunner/

├── go.mod

├── go.sum

├── main.go # Backend-сервис, точка входа

├── frontend/ # Исходники SPA-приложения

├── backend/ # HTTP-API, логика интеграций

├── microservices/

│ └── code-runner/ # Компиляция и тестирование C++-кода

├── db/ # SQL-миграции, скрипты и seed’ы

├── configs/ # Конфиги MinIO, DragonFly и др.

└── deployments/ # docker-compose.yml и Kubernetes-манифесты

* Единая модульная система. Все Go‑пакеты расположены в рамках одного модуля — это упрощает управление зависимостями и гарантирует совместимость версий.
* Разделение ответственности. Клиентская часть, API‑слой и микросервисы строго разделены, что облегчает параллельную работу команд и тестирование.
* Интеграционные сценарии. В репозитории хранятся готовые манифесты для локального запуска (Docker Compose) и продакшен‑развёртывания (Kubernetes), а также миграции для PostgreSQL.

**2.3.2 Детальная конфигурация MinIO**

MinIO используется как основное объектное хранилище. Ниже — реальный конфиг и пояснения к ключевым параметрам.

{

"version": "33",

"credential": {

"accessKey": "minioadmin",

"secretKey": "minioadmin"

},

"region": "",

"browser": "on",

"worm": "off",

"api": {

"requests\_max": 0,

"requests\_deadline": 0

},

"compression": {

"enabled": false,

"extensions": [".txt",".log",".csv",".json",".tar",".xml",".bin"],

"mime\_types":["text/csv","text/plain","application/json","application/xml"]

}

}

1. version — внутренняя версия конфига MinIO; автоматически обновляется при изменениях.
2. credential.accessKey / secretKey — учетные данные администратора; в продакшене заменяются на более сложные через Secrets.
3. region — пустая строка, т.к. локальное хранилище не имеет региональной привязки.
4. browser — включен встроенный веб‑интерфейс (:9001), что позволяет вручную проверять содержимое бакетов.
5. worm — «write once, read many» отключен, поскольку файлы могут перезаписываться при повторных отправках.
6. api.requests\_max / requests\_deadline — позволяют ограничить число одновременных запросов и таймауты на уровне MinIO API; в данной конфигурации не заданы.
7. compression — отключено, но перечислены расширения и MIME‑типы, которые при включении будут сжиматься.

Автоматическое создание бакетов происходит в коде при старте сервиса:

exists, err := client.BucketExists(ctx, bucketName)

if err != nil { /\* логируем и выходим \*/ }

if !exists {

err = client.MakeBucket(ctx, bucketName, minio.MakeBucketOptions{})

}

Таким образом, при развертывании не нужно вручную создавать бакеты code, tests и executables.

### **2.3.3 Сервис работы с MinIO: полный пример**

Представленный фрагмент кода [Приложение А] является частью пакета `services` и реализует сервис для взаимодействия с объектным хранилищем MinIO, которое используется как S3-совместимое решение для хранения файлов, таких как исходный код пользователей, тестовые данные и скомпилированные файлы. Этот сервис интегрируется в микросервисную архитектуру платформы, обеспечивая загрузку файлов и генерацию временных URL для доступа к ним. Ниже описаны ключевые компоненты и логика работы кода, а также причины, по которым он структурирован именно таким образом.

Сервис представлен в виде структуры данных, которая инкапсулирует клиент для взаимодействия с MinIO и имя бакета, используемого для хранения объектов. Это позволяет централизовать управление соединением с хранилищем и упрощает доступ к конкретному бакету, который заранее определён для хранения данных платформы (например, кода или тестов).

Цель структуры: Обеспечить единый интерфейс для операций с хранилищем, избегая повторного создания клиента MinIO для каждой операции.

Причина выбора такого подхода: Инкапсуляция клиента и имени бакета в структуре упрощает управление состоянием и повторное использование кода в разных частях приложения, а также делает код более читаемым и поддерживаемым.

Функция инициализации создаёт новый экземпляр сервиса, принимая параметры подключения: адрес конечной точки MinIO, ключи доступа (идентификатор и секретный ключ), а также имя бакета. В процессе инициализации выполняются следующие шаги:

Создаётся клиент MinIO с использованием предоставленных учётных данных и опций, где явно отключается использование SSL (для упрощения разработки или работы в локальной среде). Затем проверяется существование указанного бакета в хранилище. Если бакет не существует, он создаётся с настройками по умолчанию. В случае успешного выполнения всех шагов возвращается указатель на структуру сервиса, готовую к использованию.

Цель инициализации: Гарантировать, что сервис готов к работе с хранилищем, а бакет существует и доступен для операций.

Причина выбора такого подхода: Автоматическое создание бакета при его отсутствии упрощает настройку системы, особенно в тестовых или новых средах. Отключение SSL может быть временным решением для локальной разработки, чтобы избежать сложностей с сертификатами, хотя в продакшен-среде это должно быть включено для безопасности. Обработка ошибок на каждом этапе позволяет точно определить причину сбоя (например, неверные учётные данные или проблемы с сетью) и сообщить об этом вызывающей стороне.

Метод загрузки файлов позволяет отправить локальный файл в хранилище MinIO под указанным именем объекта. Процесс включает следующие действия:

1. Открывается локальный файл для чтения.
2. Получается информация о файле (например, его размер), необходимая для загрузки.
3. Файл загружается в указанный бакет с использованием имени объекта, при этом задаётся тип содержимого (например, "text/plain" для текстовых файлов, таких как исходный код).
4. Все ошибки, возникающие на каждом этапе (открытие файла, получение информации, загрузка), обрабатываются и возвращаются с дополнительным контекстом для упрощения диагностики.

Цель метода: Обеспечить надёжную загрузку файлов в хранилище, что критически важно для сохранения исходного кода пользователей или тестовых данных в рамках работы платформы.

Причина выбора такого подхода: Использование контекста в вызове позволяет управлять временем выполнения операции и отменять её при необходимости (например, при таймауте). Указание типа содержимого помогает клиентам, которые будут скачивать файл, правильно интерпретировать его формат. Обработка ошибок с контекстом упрощает трассировку проблем, что особенно важно в распределённой системе, где ошибка может возникнуть на любом этапе.

Метод генерации URL создаёт временную ссылку для доступа к объекту в хранилище без необходимости прямого доступа через API MinIO. Основные шаги:

1. Генерируется предварительно подписанный URL для объекта в указанном бакете с ограничением по времени действия.
2. URL возвращается в виде строки, готовой для использования.
3. Ошибки, возникающие при генерации URL, обрабатываются и возвращаются с контекстом.

Цель метода: предоставить безопасный и временный доступ к файлам, хранящимся в MinIO, без раскрытия учётных данных или предоставления постоянного доступа.

Причина выбора такого подхода: Ограничение времени действия URL (1 час) снижает риск несанкционированного доступа к файлам, что важно для защиты пользовательских данных, таких как исходный код или результаты тестов. Использование контекста позволяет управлять операцией и отменять её при необходимости. Обработка ошибок с контекстом помогает разработчикам и администраторам быстро определить причину проблемы (например, отсутствие объекта или истёкший срок действия ключа).

Код написан с учётом нескольких важных принципов, которые делают его подходящим для использования в микросервисной архитектуре обучающей платформы:

Обработка ошибок с контекстом: на каждом этапе работы (инициализация, загрузка файлов, генерация URL) ошибки оборачиваются в дополнительное описание, что упрощает их трассировку в логах системы. Это особенно важно в распределённых системах, где ошибка может возникнуть на любом уровне (сеть, файловая система, хранилище).

Использование контекста: Все операции, взаимодействующие с хранилищем, принимают контекст как параметр. Это позволяет управлять временем выполнения операций, отменять их при необходимости и передавать дополнительные метаданные, что критично для асинхронных и масштабируемых систем.

Рекомендация по именованию объектов: В комментариях к коду указано, что имена объектов в хранилище рекомендуется формировать как уникальные идентификаторы (UUID) для задач или попыток пользователей. Это предотвращает коллизии имён, когда несколько пользователей или задач могут случайно перезаписать данные друг друга.

Цель этих особенностей: обеспечить надёжность, безопасность и масштабируемость работы с хранилищем в условиях высокой нагрузки и множества параллельных операций, характерных для обучающей платформы.

Причина выбора такого подхода: Микросервисная архитектура требует чёткого управления ошибками и ресурсами, чтобы избежать узких мест и обеспечить быструю диагностику проблем. Использование UUID для имён объектов соответствует лучшим практикам работы с распределёнными хранилищами, где уникальность данных имеет первостепенное значение.

Этот сервис является частью бэкенда обучающей платформы и отвечает за взаимодействие с объектным хранилищем MinIO, которое используется для хранения файлов, необходимых для работы микросервиса CodeRunner и других компонентов системы. В частности:

* Загрузка файлов позволяет сохранять исходный код, написанный пользователями, и тестовые данные для проверки решений задач.
* Генерация временных URL обеспечивает безопасный доступ к файлам, например, для отображения результатов тестирования или скачивания скомпилированных файлов.

Такой подход к управлению файлами через S3-совместимое хранилище позволяет платформе быть масштабируемой, так как MinIO поддерживает распределённое хранение данных и высокую нагрузку, что важно для системы с большим количеством пользователей.

### **2.3.4 Формат и размещение тестовых наборов (Listing 1)**

Тестовый файл в формате JSON:

[

{ "stdin": "5 5 10", "stdout": "25\n25\n100" },

{ "stdin": "-2 2 4", "stdout": "4\n4\n16" },

{ "stdin": "100\n2000", "stdout": "10000\n4000000" }

]

* Хранится в бакете tests под именем <taskID>.json.
* Backend загружает его в MinIO перед отправкой задачи в очередь:

// Пример загрузки тестов

testsJSON, \_ := json.Marshal(tests)

err = minioService.UploadFile(ctx, "/tmp/tests.json", taskID+".json")

### **2.3.5 Публикация задач: структура и пример (Listing 2)**

Структура сообщения:

{

"id": "uuid-1234",

"codeLocation": {

"bucketName": "code",

"objectName": "uuid-1234.cpp"

},

"testsLocation": {

"bucketName": "tests",

"objectName": "uuid-1234.json"

},

"compiler": "g++"

}

* id — UUID, используется клиентом и сервисом Coderunner для корреляции.
* codeLocation — ссылка на исходник.
* testsLocation — ссылка на JSON с тестами.
* compiler — указывает команду для компиляции (зарезервировано для поддержки нескольких языков).

type Task struct {

ID string `json:"id"`

CodeLocation Location `json:"codeLocation"`

TestsLocation Location `json:"testsLocation"`

Compiler string `json:"compiler"`

}

type Location struct {

BucketName string `json:"bucketName"`

ObjectName string `json:"objectName"`

}

func PublishTask(rdb \*redis.Client, task Task) error {

data, err := json.Marshal(task)

if err != nil {

return err

}

return rdb.Publish(context.Background(), "coderunner\_task\_channel", data).Err()

}

* Redis Pub/Sub обеспечивает неблокирующую публикацию: backend сразу возвращает пользователю ответ о принятии решения.

### **2.3.6 Микросервис Coderunner: полная логика (Listing 3)**

**Инициализация и роутинг задач**

func main() {

ctx := context.Background()

redisClient := getRedisClient()

defer redisClient.Close()

minioClient := getMinioClient()

dockerClient, \_ := client.NewClientWithOpts(client.FromEnv, client.WithAPIVersionNegotiation())

var sandboxManager sandbox.Manager

if os.Getenv("USE\_TMPFS") == "true" {

sandboxManager = sandbox.NewTMPFSDockerManager(dockerClient)

} else {

sandboxManager = sandbox.NewDockerManager(dockerClient)

}

filesManager := filesctl.NewMinioManager(minioClient)

tasksToCompile := make(chan model.Task, 30)

tasksToTest := make(chan model.Task, 2)

// Запуск воркеров

for i := 0; i < 5; i++ {

go handler.HandleTasksToCompile(ctx, filesManager, sandboxManager, tasksToCompile, tasksToTest)

}

for i := 0; i < 3; i++ {

go handler.HandleTasksToTest(ctx, filesManager, sandboxManager, tasksToTest, redisClient)

}

// Подписка на канал задач

handler.HandleStartTaskCommands(ctx, redisClient, tasksToCompile)

}

* tasksToCompile: задачи на компиляцию, буфер 30.
* tasksToTest: задачи на запуск и тестирование, буфер 2.
* HandleStartTaskCommands: слушает coderunner\_task\_channel и кладёт сообщения в tasksToCompile.

Компиляция и тестирование.

1. HandleTasksToCompile:
   * Скачивает исходный файл и тестовый JSON из MinIO.
   * Вызывает sandboxManager.Compile(task) — создаёт Docker‑контейнер, монтирует исходник, выполняет g++, сохраняет бинарь.
   * Загружает бинарный файл в MinIO бакет executables.
   * Отправляет задачу в tasksToTest.
2. HandleTasksToTest:
   * Скачивает бинарь и тестовый JSON.
   * Распараллеливает проверку: запускает контейнеры или процессы Go‑горутины, которые подают stdin и сравнивают stdout.
   * Формирует []model.TestResult и публикует результат в канал coderunner\_completed\_tasks\_channel.

Пример результата:

{

"id": "uuid-1234",

"executableLocation": {

"bucketName": "executables",

"objectName": "uuid-1234.out"

},

"state": "completed",

"testsResults": [

{ "test\_id": 0, "successful": true },

{ "test\_id": 1, "successful": true },

{ "test\_id": 2, "successful": true }

]

}

* state — "completed" или "failed" при ошибках компиляции.
* testsResults — массив соответствий тестов (pass/fail).

### **2.3.7 Контейнеризация и развёртывание**

Docker Compose объединяет:

* backend: Go‑сервис REST API.
* code-runner: микросервис проверки.
* minio: хранилище S3.
* postgres: БД.
* dragonfly: Redis‑совместимый брокер.
* frontend: статические файлы SPA.

docker-compose up --build запустит все сервисы в единой сети, настроит тома и healthchecks.

Kubernetes манифесты позволяют[7]:

* Определить Deployments и Services.
* Использовать ConfigMap и Secret для конфигурации MinIO, Postgres, DragonFly.
* Настроить HPA (Horizontal Pod Autoscaler) для автоматического масштабирования воркеров.

Таким образом, GoRunner сочетает чёткое разделение ответственностей, проверенные облачные компоненты и асинхронную, масштабируемую модель обработки, что делает платформу надежным и гибким решением для автоматизации проверки пользовательских решений на C++.

## **ЗАКЛЮЧЕНИЕ**

В ходе разработки платформы GoRunner была реализована современная, отказоустойчивая и масштабируемая микросервисная архитектура, ориентированная на автоматизацию полного цикла обработки пользовательских решений программных задач. Выбор языка Go в качестве основы backend‑сервиса позволил достичь высокой производительности и эффективного использования ресурсов благодаря лёгковесным горутинам и встроенным механизмам конкурентности. Разделение кода на модули, слои сервисов и репозиториев обеспечило удобство сопровождения, повторного использования компонентов и возможности параллельной работы нескольких команд разработки.

Использование MinIO в роли S3‑совместимого объектного хранилища дало необходимые свойства — надёжность сохранения больших объёмов данных, возможность горизонтального масштабирования и прозрачный веб‑интерфейс для администрирования. Автоматическое создание бакетов, проверка и настройка прав доступа при старте сервисов упрощает процесс развёртывания и снижает вероятность ошибок в конфигурации. Генерация presigned URL для временного доступа к объектам позволяет безопасно отдавать файлы фронтенду и сторонним приложениям без прямого раскрытия учётных данных.

PostgreSQL, как надёжная реляционная база данных[8], обеспечивает целостность и согласованность метаданных — описаний задач, информации об отправках, результатах тестирования и учётных записях пользователей. Пул соединений, система миграций и правильно настроенные индексы гарантируют стабильную работу при высоких нагрузках и росте числа пользователей. Репозитории и DAO‑слой визуально и концептуально отделяют логику доступа к данным от бизнес‑процессов, что упрощает масштабирование и модификацию схемы без затрагивания кода проверяющих компонентов.

Ключевым элементом асинхронной обработки стал Redis‑совместимый сервер DragonFly, выступающий в роли брокера сообщений и кеша. Модель Pub/Sub позволила разделить приём задач на проверку и их выполнение, избавив основной сервис от блокирующих операций и обеспечив параллельную работу множества воркеров. Использование кеширования для результатов недавно проверённых отправок снизило нагрузку на базу данных и сократило время ответа фронтенду при повторных запросах.

Микросервис Coderunner, отвечающий исключительно за компиляцию и тестирование C++‑кода, демонстрирует принцип stateless‑архитектуры: каждый запрос обрабатывается независимо, без использования внутреннего состояния, что упрощает горизонтальное масштабирование и повышает отказоустойчивость. Изоляция выполнения кода в Docker‑контейнерах гарантирует безопасность и воспроизводимость среды, а встроенная поддержка tmpfs даёт гибкость в настройке производительности и использования дискового пространства.

Контейнеризация всей системы через Docker Compose[6] обеспечивает единообразие окружения разработчиков и упрощает локальную отладку. Готовые манифесты Kubernetes[7] позволяют развернуть платформу в продакшен‑кластере с учётом требований к доступности, мониторингу и автоматическому масштабированию. Использование ConfigMap и Secret для хранения конфигурации и учётных данных гарантирует безопасность при эксплуатации и облегчает управление версиями параметров.

Фронтенд, реализованный как одностраничное приложение, взаимодействует с backend через REST API, позволяя пользователям загружать решения, отслеживать статус обработки в реальном времени и просматривать подробные отчёты о тестировании. Такой подход обеспечивает высокий уровень удобства и вовлечённости пользователей, а программы на C++ проверяются в условиях, приближённых к реальным олимпиадам по программированию.

Перечисленные технологические решения — Go, MinIO, PostgreSQL, DragonFly, Docker и SPA на React — в совокупности формируют надёжную, быструю и гибкую платформу для обучения программированию. Архитектура GoRunner демонстрирует баланс между производительностью, простотой сопровождения и возможностью масштабирования, что делает её пригодной для вузовских курсов, онлайн‑олимпиад и корпоративных тренингов.

В перспективе планируется:

1. **Расширение поддержки языков.** Внедрение новых компиляторов и интерпретаторов позволит охватить широкий спектр задач на различных языках программирования.
2. **Развитие системы ролей и прав.** Добавление детального разграничения доступа для преподавателей, студентов и администраторов.
3. **Интеграция с внешними системами.** Подключение платформы к LMS, системам аналитики и облачным CI/CD.
4. **Оптимизация производительности.** Внедрение горизонтального автоскейлинга на уровне Kubernetes, доработка стратегий кеширования и балансировки нагрузки.
5. **Улучшение UX.** Расширение функционала фронтенда: визуализация статистики, интерактивные графики и система рекомендаций по улучшению кода.

Таким образом, GoRunner представляет собой современную, облачно‑ориентированную платформу, способную эффективно решать задачи автоматизированной проверки кода и служить основой для дальнейшего развития образовательных и исследовательских проектов в области программирования.

**СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ**

1. Amazon Simple Storage Service (S3) [Электронный ресурс] // Amazon Web Services. - URL: https://aws.amazon.com/ru/s3/ (дата обращения: 20.04.2025).

2. MinIO Documentation [Электронный ресурс] // MinIO. - URL: https://docs.min.io/ (дата обращения: 20.04.2025).

3. Go Programming Language Documentation [Электронный ресурс] // The Go Project. - URL: https://golang.org/doc/ (дата обращения: 20.04.2025).

4. React Documentation [Электронный ресурс] // React. - URL: https://reactjs.org/docs/ (дата обращения: 20.04.2025).

5. Dragonfly Documentation [Электронный ресурс] // Dragonfly. - URL: https://www.dragonflydb.io/docs (дата обращения: 20.04.2025).

6. Docker Documentation [Электронный ресурс] // Docker. - URL: https://docs.docker.com/ (дата обращения: 20.04.2025).

7. Kubernetes Documentation [Электронный ресурс] // Kubernetes. - URL: https://kubernetes.io/docs/ (дата обращения: 20.04.2025).

8. PostgreSQL Documentation [Электронный ресурс] // PostgreSQL. - URL: https://www.postgresql.org/docs/ (дата обращения: 20.04.2025).

9. Redis Documentation [Электронный ресурс] // Redis. - URL: https://redis.io/documentation (дата обращения: 20.04.2025).

10. Bass, L. Software Architecture in Practice / L. Bass, P. Clements, R. Kazman. - 3rd Edition. - Addison-Wesley, 2012. - 624 с.

# **ПРИЛОЖЕНИЕ А. Сервис работы с MinIO: полный пример**

package services

import (

"context"

"fmt"

"os"

"github.com/minio/minio-go/v7"

"github.com/minio/minio-go/v7/pkg/credentials"

)

// MinioService управляет загрузкой и выдачей объектов из MinIO

type MinioService struct {

client \*minio.Client

bucketName string

}

// NewMinioService инициализирует клиента и проверяет бакет

func NewMinioService(endpoint, accessKey, secretKey, bucketName string) (\*MinioService, error) {

client, err := minio.New(endpoint, &minio.Options{

Creds: credentials.NewStaticV4(accessKey, secretKey, ""),

Secure: false,

})

if err != nil {

return nil, fmt.Errorf("создание клиента MinIO: %w", err)

}

ctx := context.Background()

exists, err := client.BucketExists(ctx, bucketName)

if err != nil {

return nil, fmt.Errorf("проверка бакета: %w", err)

}

if !exists {

if err = client.MakeBucket(ctx, bucketName, minio.MakeBucketOptions{}); err != nil {

return nil, fmt.Errorf("создание бакета: %w", err)

}

}

return &MinioService{client: client, bucketName: bucketName}, nil

}

// UploadFile читает локальный файл и загружает его в MinIO

func (s \*MinioService) UploadFile(ctx context.Context, filePath, objectName string) error {

file, err := os.Open(filePath)

if err != nil {

return fmt.Errorf("открытие файла: %w", err)

}

defer file.Close()

info, err := file.Stat()

if err != nil {

return fmt.Errorf("получение FileInfo: %w", err)

}

\_, err = s.client.PutObject(ctx, s.bucketName, objectName, file, info.Size(),

minio.PutObjectOptions{ContentType: "text/plain"})

if err != nil {

return fmt.Errorf("PutObject: %w", err)

}

return nil

}

// GetFileURL генерирует временный URL для объекта (1 час)

func (s \*MinioService) GetFileURL(ctx context.Context, objectName string) (string, error) {

url, err := s.client.PresignedGetObject(ctx, s.bucketName, objectName, 3600, nil)

if err != nil {

return "", fmt.Errorf("PresignedGetObject: %w", err)

}

return url.String(), nil

}