|  |  |
| --- | --- |
|  | **Министерство науки и высшего образования Российской Федерации**  **Федеральное государственное бюджетное образовательное учреждение**  **высшего образования**  **«Московский государственный технический университет**  **имени Н.Э. Баумана**  **(национальный исследовательский университет)»**  **(МГТУ им. Н.Э. Баумана)** |

ФАКУЛЬТЕТ \_\_\_\_\_\_\_ИНФОРМАТИКА И СИСТЕМЫ УПРАВЛЕНИЯ\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

КАФЕДРА \_\_\_\_\_\_\_\_\_СИСТЕМЫ ОБРАБОТКИ ИФОРМАЦИИ И УПРАВЛЕНИЯ\_\_\_\_\_\_\_\_\_

**РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

***К НАУЧНО-ИССЛЕДОВАТЕЛЬСКОЙ РАБОТЕ***

***НА ТЕМУ:***

***Анализ разработки и развертывания “Многофункциональной информационной системы эволюции развития и современного состояния ракетно-космической отрасли” \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***

***\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***

***\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***

***\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***

Студент \_\_\_ИУ5-33М\_\_\_\_ **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_**А.Д. Калашников**\_\_**

(Группа) (Подпись, дата) (И.О.Фамилия)

Руководитель **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_**Ю.Е. Гапанюк **\_\_\_\_\_**

(Подпись, дата) (И.О.Фамилия)

Консультант **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

(Подпись, дата) (И.О.Фамилия)

*2024 г.*

**Министерство науки и высшего образования Российской Федерации**

**Федеральное государственное бюджетное образовательное учреждение**

**высшего образования**

**«Московский государственный технический университет имени Н.Э. Баумана**

**(национальный исследовательский университет)»**

**(МГТУ им. Н.Э. Баумана)**

УТВЕРЖДАЮ

Заведующий кафедрой \_\_\_ИУ5\_\_

(Индекс)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_В.И. Терехов\_\_

(И.О.Фамилия)

« \_\_\_\_\_ » \_\_\_\_\_\_\_\_\_\_\_\_ 20 \_\_\_\_ г.

**ЗАДАНИЕ**

**на выполнение научно-исследовательской работы**

по теме \_\_ Анализ разработки и развертывания “Многофункциональной информационной системы эволюции развития и современного состояния ракетно-космической отрасли”

Студент группы \_\_\_ИУ5-33М\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Калашников Артем Дмитриевич\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(Фамилия, имя, отчество)

Направленность НИР (учебная, исследовательская, практическая, производственная, др.)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_учебная\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Источник тематики (кафедра, предприятие, НИР) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

График выполнения НИР: 25% к \_5\_ нед., 50% к \_9\_ нед., 75% к \_13\_ нед., 100% к \_17\_ нед.

***Техническое задание \_\_ Провести анализ существующих решений и технологий для разработки информационной системы. Изучить архитектурные подходы, программные решения для серверной части и клиентской части, базы данных, а также инструменты аналитики и визуализации. Проанализировать методы контейнеризации и оркестрации, а также системы развертывания и облачные платформы. Результатом станет отчет с анализом существующих решений, сравнением технологий и рекомендациями для развертывания системы. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***

***Оформление научно-исследовательской работы:***

Расчетно-пояснительная записка на \_\_36\_\_ листах формата А4.

Перечень графического (иллюстративного) материала (чертежи, плакаты, слайды и т.п.)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Дата выдачи задания «09» \_\_сентября\_\_ 2024 г.

**Руководитель НИР**  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_Ю.Е. Гапанюк \_\_\_\_

(Подпись, дата) (И.О.Фамилия)

**Студент \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_**А.Д. Калашников**\_\_**

(Подпись, дата) (И.О.Фамилия)

Примечание: Задание оформляется в двух экземплярах: один выдается студенту, второй хранится на кафедре.

[Введение 4](#_Toc186180885)

[Архитектура 4](#_Toc186180886)

[1. Требования к архитектуре системы 4](#_Toc186180887)

[2. Архитектурные решения и стек технологий 6](#_Toc186180888)

[3. Настройка серверной инфраструктуры 7](#_Toc186180889)

[4. Особенности проектирования 9](#_Toc186180890)

[5. Проектирование и масштабирование 9](#_Toc186180891)

[6. Мониторинг и управление 12](#_Toc186180892)

[Управление и поддержка системы 14](#_Toc186180893)

[Сравнение микросервисной архитектуры и монолита 15](#_Toc186180894)

[Сравнение языков программирования серверной части 18](#_Toc186180895)

[Сравнение баз данных 23](#_Toc186180896)

[Остальные инструменты разработки 28](#_Toc186180897)

[Заключение 33](#_Toc186180898)

[Архитектурные решения и технологический выбор 34](#_Toc186180899)

[Инфраструктура и автоматизация процессов 34](#_Toc186180900)

[Практическая значимость и перспективы 35](#_Toc186180901)

[Дальнейшие направления развития: 35](#_Toc186180902)

[Список литературы 36](#_Toc186180903)

Введение

Современная Россия обладает огромными возможностями для успешного развития космической деятельности с учетом меняющихся условий как внутри страны, так и на мировой арене. Уникальный потенциал космической техники, многолетний опыт эксплуатации космических комплексов, участие в международном сотрудничестве по исследованию и практическому использованию космического производства в сочетании с космической инфраструктурой и высококвалифицированными кадрами составляют тот прочный фундамент, на основе которого Россия укрепит позиции в мировой космонавтике. Многофункциональная информационная система эволюции развития и современного состояния ракетно-космической отрасли представляет собой набор информационно-технологических подсистем, функционирующих, как единый программный комплекс. В результате работы были выделены следующие подсистемы: интерактивный навигатор по ракетно-космическим адресам России, космическая энциклопедия, объекты космической инфраструктуры, виртуальный музей и образовательные возможности наследия и современного состояния ракетно-космической области. Данная система предназначена для ознакомления и изучения истории и объектов космического наследия, просмотр ракетно-космических объектов.

Архитектура

1. Требования к архитектуре системы

При разработке многофункциональной информационной системы для ракетно-космической отрасли необходимо учитывать как функциональные, так и нефункциональные требования. Важным аспектом является обеспечение гибкости, масштабируемости, надежности и безопасности системы. Основные требования к архитектуре системы включают:

* **Модульность**

Архитектура должна быть построена на основе модульного подхода, что позволяет добавлять новые компоненты и функции без нарушения работы всей системы.

Модули системы должны быть независимыми и легко интегрироваться с другими подсистемами.

* **Масштабируемость**

Система должна поддерживать горизонтальное и вертикальное масштабирование для увеличения производительности и объема обрабатываемых данных.

Возможность адаптации к увеличению нагрузки при росте числа пользователей и объема информации.

* **Интеграция с внешними системами**

Поддержка взаимодействия с существующими решениями, такими как системы управления проектами (PLM, ERP), базы данных и аналитические платформы.

Наличие API для интеграции с другими информационными системами.

* **Обеспечение безопасности**

Защита данных на всех этапах – от передачи до хранения, включая использование шифрования и контроль доступа.

* **Высокая доступность и отказоустойчивость**

Архитектура должна обеспечивать минимальные простои за счет использования резервирования компонентов, балансировки нагрузки и систем мониторинга.

Поддержка горячего восстановления в случае сбоев.

* **Гибкость и адаптивность**

Возможность настройки системы под уникальные потребности конкретного предприятия.

Поддержка различных рабочих процессов и сценариев использования.

* **Поддержка обработки больших данных**

Система должна быть способна обрабатывать большие объемы структурированных и неструктурированных данных, включая телеметрию, инженерные расчеты и аналитические отчеты.

Интеграция с платформами для анализа данных и машинного обучения.

* **Удобство использования**

Интуитивно понятный пользовательский интерфейс, обеспечивающий легкий доступ к функционалу.

Поддержка различных платформ и устройств, включая настольные компьютеры и мобильные устройства.

* **Использование современных технологий**

Применение микросервисной архитектуры или архитектуры, основанной на событиях, для повышения гибкости и управляемости.

Использование облачных технологий и контейнеризации (например, Kubernetes, Docker) для упрощения развертывания и управления системой.

* **Поддержка регламентов и стандартов отрасли**

Соответствие требованиям ГОСТ и международных стандартов, связанных с проектированием.

Встроенные инструменты для документирования процессов и соблюдения нормативных актов.

Эти требования необходимы для разработки качественной и надежной информационной системы, способной удовлетворить потребности ракетно-космической отрасли, повышая её эффективность и конкурентоспособность.

1. Архитектурные решения и стек технологий

Исходя из требований к архитектуре системы, были приняты архитектурные решения и выбран стек технологий, обеспечивающий надежность, гибкость и масштабируемость.

* **PostgreSQL** и **Redis** для работы с большими объемами сложных структурированных данных и кэширования и ускорения операций;
* **Golang** как высокопроизводительный и безопасный язык разработки. Система разделена на отдельные микросервисы, каждый из которых отвечает за свою функциональность. Микросервисы взаимодействуют друг с другом через REST API и асинхронные очереди (RabbitMQ);
* **Docker** и **Kubernetes** для контейнеризации и оркестрации. Это позволяет изолировать компоненты и минимизировать проблемы совместимости, а также обеспечивает их масштабируемости и автоматическое восстановление в случае отказа;
* **GitLab** и **GitLab CI/CD** для хранения и управления кода и для автоматизации процессов тестирования, сборки и развертывания. Это обеспечивает быстрое и надежное обновление системы;
* **Grafana** и **Prometheus** для мониторинга производительности;
* **Elasticsearch** и **Kibana** для анализа логов.
* **Nginx** настроен как обратный прокси-сервер для балансировки нагрузки между микросервисами и обеспечения доступа к фронтенд- и бэкенд-компонентам;

1. Настройка серверной инфраструктуры

Для развертывания всей системы работа началась с установки Ubuntu Server, которая послужила базой для дальнейшей конфигурации. Сразу после установки операционной системы сервер был обновлён: обновлены все системные пакеты, чтобы избежать проблем с устаревшими компонентами. Затем была произведена настройка swap-файла — изначально создан файл по умолчанию, но он оказался недостаточным для предполагаемой нагрузки. Мы увеличили его размер до необходимых параметров, а также внесли изменения в системные параметры для правильной работы. Одновременно отключили гибернацию, поскольку для серверных задач она неактуальна и даже может мешать.

Для обеспечения удалённого администрирования сервер был подготовлен к работе через SSH. Настройка включала установку и настройку OpenSSH, добавление ключей доступа, а также защиту подключения с помощью брандмауэра. После этого на сервер были установлены базовые инструменты: Git, wget, и ряд других утилит, необходимых для загрузки, сборки и конфигурирования программного обеспечения.

Чтобы обеспечить среду для разработки и выполнения современных приложений, на сервер были установлены linuxbrew, nvm, Node.js и GoLang. Эти инструменты предоставляют возможность компиляции, управления версиями языков программирования и работы с библиотеками.

На следующем этапе был установлен и настроен Docker, который стал ключевым компонентом для контейнеризации приложений. Сразу после Docker приступили к установке и конфигурированию Kubernetes — одной из главных систем для оркестрации контейнеров. Для отладки локальных кластеров дополнительно установили Kind и Minikube, что значительно упростило работу с тестовыми средами. Для управления приложениями на Kubernetes был установлен Helm, что позволило быстро развертывать готовые шаблоны приложений и управлять их настройками. После этого была выполнена инициализация кластера Kubernetes, включающая развертывание основных компонентов и базовой инфраструктуры.

Затем началась работа с веб-сервером. Установили и настроили Nginx, который был конфигурирован в качестве обратного прокси-сервера. Настройки позволили управлять запросами, распределять нагрузку и защищать внутренние сервисы от внешнего воздействия.

Далее на сервере развернули GitLab, который стал центром управления репозиториями и процессами CI/CD. После установки была произведена его настройка: создана конфигурация для работы через Nginx, подключение SSL и настройка хранилищ. Также добавили GitLab Runner для выполнения CI/CD пайплайнов. Конфигурация пайплайнов включала автоматическую сборку, тестирование и деплой приложений.

Работа с базами данных началась с установки PostgreSQL и Redis. Эти базы данных были настроены с учётом требований проекта: созданы роли, настроены базы данных с необходимыми правами доступа. Для удобного развертывания баз данных был разработан docker-compose файл, который включал не только стандартную установку, но и дополнительную настройку, например, изменения параметров хранения и подключения.

Таким образом, сервер был подготовлен как полноценная платформа для разработки, тестирования и развертывания приложений. Все инструменты и технологии были интегрированы в единую инфраструктуру, обеспечивающую стабильность и масштабируемость.

1. Особенности проектирования

API Gateway: используется для объединения всех микросервисов в единое точку доступа.

Мониторинг и логирование: Prometheus и Grafana интегрированы для мониторинга системы, а ELK (Elasticsearch, Logstash, Kibana) для централизованного логирования.

Тестирование: Тесты покрывают все уровни системы — юнит-тесты, интеграционные тесты и тесты нагрузки.

Безопасность: Настроена двухфакторная аутентификация и шифрование конфиденциальных данных.

Такой подход к проектированию и выбору технологий позволяет удовлетворить все требования системы, обеспечивая её надежность, масштабируемость и производительность.

1. Проектирование и масштабирование

На этапе проектирования были учтены особенности модели данных, включая взаимосвязи сущностей и требования к интеграции с другими системами. Оптимизация запросов и индексов повышает скорость работы базы данных. На рисунке 1 представлено взаимодействие пользователя с системой. Показана связь микросервисов и таблиц базы данных. Было принято решение об определении общего формата для данных, которые, в дальнейшем, по мапингу будут собираться в полноценную структуру данных, со всей необходимой информацией относительно типа. На рисунке 2 представлена диаграмма связей и сущностей базы данных. В нашей системе мы выделили 4 основных типа данных, а именно: Люди, События, Музеи и Объекты космической инфраструктуры. Общая таблица Data несет в себе основные атрибуты и тип данных, в зависимости от которого мы определяем из какой таблицы дать информационные поля.

Такой подход нам позволяет легко масштабировать систему, а также открывать возможность к горизонтальному масштабированию, добавление новых микросервисов(подсистем) и типов данных. Данная архитектура позволяет безболезненно добавлять новые подсистемы, которые будут отлично взаимодействовать между собой.

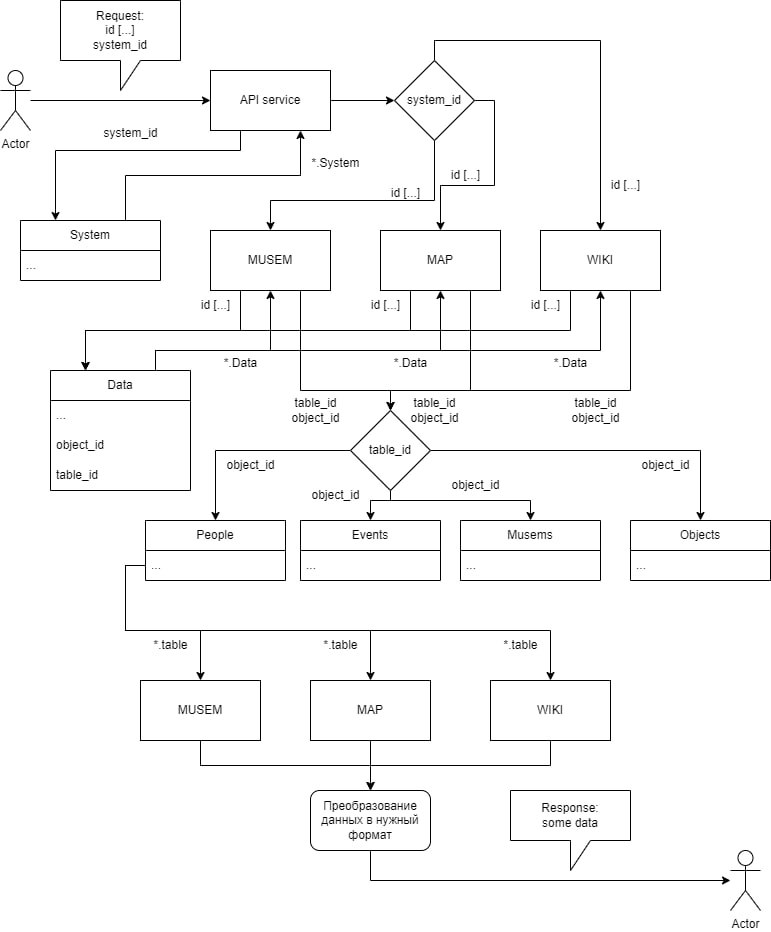


Рисунок 1. Диаграмма взаимодействия пользователя с системой

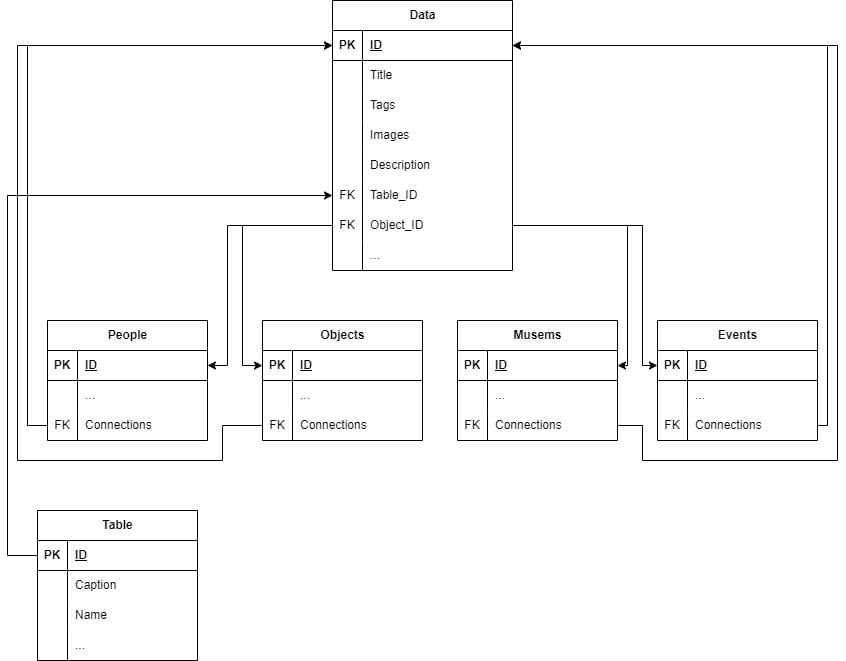


Рисунок 2. Диаграмма таблиц базы данных

Система поддерживает горизонтальное масштабирование, позволяя добавлять узлы для увеличения производительности. Оркестрация на Kubernetes автоматизирует управление нагрузкой, что делает её устойчивой к пиковым нагрузкам и сбоям.

1. Мониторинг и управление

Для обеспечения надежности и стабильной работы многофункциональной информационной системы необходимо внедрить систему мониторинга, управления логированием, алертингом и управлением ресурсами. Это позволит своевременно выявлять и устранять проблемы, а также эффективно использовать ресурсы.

* Цели мониторинга и управления

Непрерывный мониторинг:

Сбор данных о состоянии всех компонентов системы (нагрузка на CPU, использование памяти, состояние сети, доступность сервисов).

Управление логированием:

Централизация логов из всех микросервисов для упрощения анализа и устранения ошибок.

Анализ производительности

Выявление узких мест в производительности приложений.

Алертинг

Настройка уведомлений для команды на случай сбоев или отклонений от нормы.

* Инструменты мониторинга и управления

Для обеспечения полноценного мониторинга и управления инфраструктурой был выбран набор инструментов, каждый из которых выполняет свою функцию и позволяет отслеживать состояние системы, анализировать производительность и реагировать на возникающие проблемы:

|  |  |  |
| --- | --- | --- |
| **Компонент** | **Инструмент** | **Функция** |
| Мониторинг инфраструктуры | Prometheus + Node Exporter | Сбор метрик с серверов и контейнеров |
| Визуализация метрик | Grafana | Настройка дашбордов для анализа состояния |
| Логирование | ELK Stack (Elasticsearch, Logstash, Kibana) | Сбор, хранение и анализ логов |
| Алертинг | Alertmanager | Уведомления на основе метрик Prometheus |
| Мониторинг контейнеров | cAdvisor | Мониторинг использования ресурсов контейнеров |
| Управление логами в Kubernetes | Fluentd | Сбор логов из подов Kubernetes |

Таблица 1. Инструменты для обеспечения мониторинга и управления инфраструктурой

Начали с установки и конфигурации Prometheus, который выступает в качестве системы сбора метрик. Основное внимание уделили настройке сбора данных с серверов с помощью Node Exporter и мониторинга ресурсов контейнеров через cAdvisor. Prometheus был развернут в Docker-контейнере, и в его конфигурации определены задания для сбора метрик от этих компонентов.

Следующим шагом была настройка визуализации метрик с использованием Grafana. Этот инструмент позволяет создавать дашборды, наглядно отображающие состояние системы. В качестве источника данных Grafana был подключен к Prometheus. Для удобства использовали готовые шаблоны дашбордов из открытых библиотек, адаптировав их под текущие задачи.

Для мониторинга событий, требующих оперативной реакции, был установлен Alertmanager. Он был интегрирован с Prometheus для обработки триггеров, возникающих на основе заранее настроенных правил, таких как превышение пороговых значений нагрузки на CPU или памяти. Настроили отправку уведомлений по электронной почте, что обеспечивает быструю реакцию команды на инциденты.

Система логирования была организована с помощью ELK Stack, который включает Elasticsearch для хранения данных, Logstash для обработки логов и Kibana для их визуализации. Эта система была развернута в контейнерах и сконфигурирована для интеграции с остальными компонентами. Логи собирались с серверов и приложений, обеспечивая полный цикл управления информацией.

Дополнительно для сбора логов из подов Kubernetes был установлен Fluentd. Этот инструмент интегрирован с системой Kubernetes через Helm и направляет логи в Logstash для дальнейшей обработки.

Настроенные инструменты обеспечили не только сбор данных, но и их анализ, визуализацию и оперативное уведомление команды о критических событиях. Такая система позволяет не только эффективно управлять инфраструктурой, но и заранее предотвращать возможные сбои.

Управление и поддержка системы

Мониторинг состояния

Grafana для анализа метрик и состояния системы.

Настройка алерты для оперативного реагирования на проблемы.

Регулярное обслуживание

Проверка журналов (Kibana) для анализа ошибок.

Обновления инструментов (Docker, Kubernetes, Prometheus).

Тестирование производительности

Использование инструментов, таких как locust или k6, для проведения нагрузочного тестирования.

Резервное копирование

Настройка автоматического резервного копирование базы данных и конфигураций через Cron или Kubernetes Jobs.

Основания выбора архитектурных решений

Сравнение микросервисной архитектуры и монолита

Микросервисная архитектура и монолит представляют собой два основных подхода к построению программных систем. Их выбор зависит от характера проекта, требований к масштабируемости, сложности поддержки и множества других факторов. Рассмотрим основные различия, преимущества и недостатки.

|  |  |  |
| --- | --- | --- |
| **Критерий** | **Микросервисы** | **Монолит** |
| **Масштабируемость** | Горизонтальная: масштабируется отдельно каждый сервис | Вертикальная: масштабируется всё приложение целиком |
| **Независимость разработки** | Высокая: команды могут работать автономно | Низкая: любые изменения требуют согласования |
| **Производительность** | Зависит от взаимодействия между сервисами (иногда ниже) | Высокая, благодаря локальным вызовам |
| **Сложность разработки** | Высокая: требуется настройка коммуникации и управления | Относительно низкая |
| **Обновления** | Могут быть сделаны отдельно для каждого сервиса | Требуется развертывание всего приложения |
| **Отказоустойчивость** | Выше: отказ одного сервиса не приводит к падению всей системы | Ниже: ошибка может привести к краху всего приложения |
| **Потребление ресурсов** | Выше из-за множества отдельных сервисов | Меньше, так как приложение работает в одном процессе |
| **Требования к DevOps** | Высокие: сложные процессы CI/CD и мониторинга | Низкие, достаточно простого пайплайна |
| **Время развертывания** | Длиннее, так как требуется управление каждым сервисом | Короткое, единое развертывание |

Таблица 2. Сравнение монолита и микросервисной архитектуры

1. Производительность
   * Микросервисы: В среднем добавляют ~5-10 мс задержки на межсервисное взаимодействие через REST или gRPC. Это делает систему чуть медленнее, но даёт гибкость в обработке запросов.
   * Монолит: Локальные вызовы обеспечивают минимальные задержки (~1 мс), что делает монолит более производительным в плане обработки транзакций.
2. Масштабируемость
   * В микросервисах масштабирование происходит точечно. Например, если у нас есть 10 сервисов, можно увеличить мощность одного из них для обработки возросшей нагрузки, сократив затраты.
   * Монолит требует масштабирования всего приложения, даже если только один модуль перегружен. Это может увеличить затраты в 2-3 раза при пиковых нагрузках.
3. Ресурсы
   * Микросервисы: При 10 сервисах каждое приложение может использовать в среднем 0.5-1 ГБ оперативной памяти, что приводит к общему потреблению 5-10 ГБ.
   * Монолит: Монолитное приложение в аналогичных условиях будет использовать 3-5 ГБ памяти.
4. Командная работа
   * В случае микросервисов команды могут параллельно разрабатывать и развертывать сервисы, сокращая среднее время выпуска новой функции на ~20-30%.
   * Монолит требует последовательной разработки, что увеличивает время релиза.

**Плюсы и минусы**

**Микросервисы**

Преимущества:

* Независимость команд разработки.
* Возможность точечного масштабирования.
* Устойчивость к отказам отдельных компонентов.
* Быстрота обновлений: развертывание отдельных сервисов.

Недостатки:

* Увеличение сложности инфраструктуры.
* Высокие требования к мониторингу, DevOps и CI/CD.
* Дополнительная задержка из-за сетевого взаимодействия.

**Монолит**

Преимущества:

* Простота разработки и развертывания.
* Лучшая производительность за счёт локальных вызовов.
* Меньше требований к DevOps.

Недостатки:

* Ограниченная масштабируемость.
* Уязвимость: ошибка в одном модуле может обрушить всю систему.
* Долгий цикл обновлений и тестирования.

После детального анализа и учёта численных показателей было принято решение выбрать микросервисную архитектуру. Этот выбор объясняется следующими причинами:

Масштабируемость: Микросервисы позволяют эффективно масштабировать только те компоненты, которые подвержены высокой нагрузке. Это снижает расходы на серверные мощности в условиях роста системы.

Независимость разработки: Возможность разделить проект на независимые сервисы, которыми управляют отдельные команды, значительно ускоряет разработку новых функций и улучшает контроль качества.

Гибкость в обновлениях: Микросервисы можно обновлять и развертывать по отдельности, что сокращает время простоя и позволяет быстро реагировать на изменения.

Устойчивость: Даже в случае отказа одного из сервисов, остальные продолжают работать, что минимизирует вероятность полной остановки системы.

Микросервисная архитектура подходит для сложных, масштабируемых и динамичных систем, таких как наша, где важно быстро адаптироваться к изменениям, эффективно распределять ресурсы и обеспечивать устойчивость к сбоям.

Сравнение языков программирования серверной части

Для бэкенд-разработки существует множество языков программирования, каждый из которых имеет свои особенности. Рассмотрим несколько популярных языков для бэкенда, включая Go (Golang), и проведём числовое сравнение по ключевым параметрам: эффективность, скорость, безопасность, качество кода, простота, распространенность на рынке.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Параметр** | **Go (Golang)** | **Python** | **Java** | **Ruby** | **Node.js (JavaScript)** |
| **Скорость выполнения** | Очень высокая | Средняя (4-5x медленнее Go) | Высокая | Средняя (4-5x медленнее Go) | Средняя (2-3x медленнее Go) |
| **Потребление памяти** | Низкое | Высокое | Среднее | Среднее | Среднее |
| **Безопасность типов** | Строгая типизация | Динамическая | Строгая типизация | Динамическая | Динамическая |
| **Качество кода** | Высокое, простота | Среднее (порог ошибок высокий) | Высокое | Среднее | Среднее |
| **Простота** | Высокая | Очень высокая | Средняя | Средняя | Высокая |
| **Популярность на рынке** | Растёт быстро | Очень высокая | Высокая | Средняя | Очень высокая |
| **Поддержка многозадачности** | Отличная (параллельное выполнение) | Хорошая (через потоки) | Отличная (потоки) | Хорошая (потоки) | Хорошая (потоки) |
| **Поддержка фреймворков** | Ограниченная, но развивающаяся | Очень высокая | Очень высокая | Высокая | Очень высокая |

Таблица 3. Сравнение языков программирования серверной части

Скорость выполнения

* Go (Golang): Go является одним из самых быстрых языков для серверной разработки. Это компилируемый язык с низким уровнем абстракции, что даёт ему преимущество в скорости выполнения. Go может обрабатывать 1,000,000 запросов в секунду при правильной настройке.
* Python: Python примерно в 4-5 раз медленнее Go при обработке идентичных запросов. В реальных тестах на выполнение 1,000,000 запросов в секунду Python может обрабатывать ~200,000-250,000 запросов.
* Java: Java быстрее Python, но медленнее Go. Java позволяет обрабатывать около 600,000 запросов в секунду при настройке JVM для оптимальной работы.
* Ruby: Ruby примерно в 4-5 раз медленнее Go. Он обрабатывает около 150,000-200,000 запросов в секунду.
* Node.js (JavaScript): Node.js находится между Python и Go, обрабатывая около 400,000-500,000 запросов в секунду.

Потребление памяти

* Go: Go известен своим низким потреблением памяти. Обычно для обработки одного запроса ему требуется около 1-5 МБ памяти, что делает его оптимальным для систем с высоким трафиком.
* Python: Python требует значительно больше памяти — от 10 до 15 МБ на запрос, что делает его менее эффективным по сравнению с Go в контексте серверных приложений с высокой нагрузкой.
* Java: Java использует значительно больше памяти, поскольку работает на виртуальной машине (JVM). Обычно для одного запроса Java требует около 10-20 МБ.
* Ruby: Ruby потребляет от 10 до 15 МБ на запрос, что незначительно хуже по сравнению с Python и Java.
* Node.js: Node.js имеет относительно низкое потребление памяти, но несколько выше, чем у Go, с диапазоном около 5-8 МБ на запрос.

Безопасность типов

* Go: Строгая типизация в Go помогает предотвращать типичные ошибки на стадии компиляции, что повышает безопасность кода и снижает вероятность возникновения ошибок в производственной среде.
* Python: Динамическая типизация в Python снижает безопасность и может приводить к ошибкам, которые проявляются только во время выполнения.
* Java: Строгая типизация в Java аналогична Go, но Java может быть менее гибкой, что увеличивает сложность кода.
* Ruby: Также использует динамическую типизацию, что снижает безопасность, особенно в крупных проектах.
* Node.js: JavaScript также динамично типизируемый язык, что делает его менее безопасным по сравнению с Go и Java.

Простота разработки

* Go: Очень прост в освоении и использовании. Минималистичный синтаксис и строгость языка помогают разработчикам писать чистый и понятный код. Простой подход к параллелизму (goroutines) ускоряет разработку многозадачных приложений.
* Python: Один из самых простых языков для разработки благодаря читаемому синтаксису. Однако для реализации многозадачности и параллелизма требуется больше усилий.
* Java: Требует больше времени для изучения, особенно из-за большого количества фреймворков и настроек. Однако, он поддерживает параллельные вычисления и многозадачность через потоки.
* Ruby: Простой и удобный для разработчиков, однако его синтаксис и архитектурные особенности могут быть менее понятными в долгосрочной перспективе, особенно для больших приложений.
* Node.js: Благодаря асинхронной модели и использованию JavaScript, Node.js является относительно простым для новичков, особенно для тех, кто уже знаком с этим языком.

Популярность на рынке

* Go: Несмотря на свою относительную молодость, Go стремительно набирает популярность, особенно в области микросервисов и высоконагруженных систем. В 2023 году Go стал одним из самых востребованных языков на таких платформах, как GitHub и Stack Overflow.
* Python: Один из самых популярных языков на рынке, особенно в области науки о данных и веб-разработки. Однако его скорость и производительность ограничивают его использование для высоконагруженных бэкендов.
* Java: Всё ещё широко используется в корпоративных приложениях и финансовых системах, но теряет популярность среди стартапов в пользу более лёгких языков, таких как Go или Node.js.
* Ruby: В последние годы Ruby теряет популярность, в основном используется в небольших стартапах и проектах, требующих быстрой разработки.
* Node.js: Один из самых популярных инструментов для создания бэкендов, особенно в стартапах и для приложений с высокой асинхронной нагрузкой.

Go (Golang) представляет собой высокоэффективный и производительный язык программирования, который идеально подходит для разработки бэкенд-решений, особенно в условиях высоконагруженных и масштабируемых систем. Одной из его ключевых особенностей является исключительная скорость выполнения. В сравнении с другими популярными языками, такими как Python, Ruby и Node.js, Go демонстрирует значительно лучшие результаты по производительности, что особенно важно в тех случаях, когда приложения должны обрабатывать большое количество запросов в секунду, обеспечивая минимальную задержку.

Кроме того, Go эффективен в плане использования системных ресурсов, что позволяет разрабатывать решения, способные обрабатывать большие объёмы трафика при ограниченных вычислительных мощностях. Это делает его идеальным выбором для создания высоконагруженных сервисов и облачных инфраструктур, где каждое затрачиваемое на выполнение операции миллисекунду или килобайт памяти критичен.

Не менее важной характеристикой Go является строгая статическая типизация, которая существенно повышает безопасность кода. В отличие от динамически типизированных языков, таких как Python, Ruby и JavaScript, Go позволяет обнаружить потенциальные ошибки на стадии компиляции, что значительно снижает количество багов в процессе разработки и эксплуатации. Это важный фактор для создания надежных и стабильных приложений, особенно в критичных для бизнеса системах.

С точки зрения разработки Go отличается минималистичным синтаксисом и простотой освоения, что способствует высокой производительности разработчиков. Язык был специально спроектирован с акцентом на параллелизм, и поддержка многозадачности через механизм goroutines позволяет эффективно распределять нагрузку между доступными ядрами процессора, не создавая излишней сложности в коде. Это делает Go превосходным выбором для разработки распределённых систем и микросервисной архитектуры, где требования к масштабируемости и параллелизму стоят на первом месте.

Наконец, несмотря на то, что Go пока не столь широко распространён на рынке, как Python или Java, его популярность продолжает расти, особенно в контексте разработки высокопроизводительных приложений. Сильное сообщество разработчиков, активное развитие экосистемы и широкая поддержка от крупных компаний, таких как Google, делают Go перспективным языком для будущего, особенно в сфере разработки облачных решений и микросервисов.

Таким образом, Go (Golang) является оптимальным выбором для бэкенд-разработки, когда приоритетами становятся высокая производительность, эффективное использование ресурсов, безопасность кода и масштабируемость системы.

Сравнение баз данных

Для сравнения баз данных были рассмотрены несколько популярных вариантов, подходящих для различных типов приложений: PostgreSQL, MySQL, MongoDB и Redis. Они широко используются в индустрии, и их выбор зависит от требований проекта. Сравним их по основным характеристикам.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Параметр** | **PostgreSQL** | **MySQL** | **MongoDB** | **Redis** |
| **Тип СУБД** | Реляционная | Реляционная | Документо-ориентированная | Ключ-значение |
| **Язык запросов** | SQL (расширенный) | SQL | BSON (Mongo Query) | Нет, команды |
| **Поддержка ACID** | Полная | Полная | Частичная | Нет |
| **Производительность** | Высокая на сложных запросах | Высокая для чтения и записи | Высокая на больших объемах данных | Максимальная, минимальная задержка |
| **Масштабируемость** | Горизонтальная и вертикальная | В основном вертикальная | Горизонтальная | Горизонтальная |
| **Хранилище** | Реляционные таблицы | Реляционные таблицы | JSON-документы | В оперативной памяти |
| **Поддержка индексов** | Разнообразные типы индексов | Основные индексы | Полнотекстовые, геопространственные | Нет традиционных индексов |
| **Потребление памяти** | Умеренное | Низкое | Высокое для больших объемов | Зависит от объема данных |
| **Сценарии использования** | Комплексные транзакции, аналитика | Веб-приложения, CMS | NoSQL-приложения, Big Data | Кэширование, очереди |
| **Популярность** | Высокая | Высокая | Средняя | Высокая |

Таблица 4. Сравнение основных баз данных

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Параметр | PostgreSQL | MySQL | MongoDB | Redis |
| Время выполнения SELECT (мс) | ~1.5 | ~1.2 | ~1.8 | <0.1 |
| Время выполнения INSERT (мс) | ~2.0 | ~1.8 | ~2.5 | <0.1 |
| Операции в секунду (RPS) | ~20000 | ~25,000~25000 | ~18000~18000 | ~80000 |
| Пиковая нагрузка (RPS) | ~500000 | ~600,000 | ~400,000 | ~1,000,000 |
| Потребление памяти (МБ) | ~200 | ~150 | ~300 | Зависит от данных (~100-2000) |

Таблица 5. Сравнение производительности баз данных

**Ключевые показатели**

* Производительность:
  + PostgreSQL показывает лучшие результаты при сложных аналитических запросах и обработке транзакций благодаря продвинутым возможностям SQL.
  + MySQL быстрее на простых операциях чтения-записи, но уступает PostgreSQL в обработке сложных запросов.
  + MongoDB превосходит по скорости работы с большими массивами данных, когда требуется гибкость хранения.
  + Redis обеспечивает минимальную задержку благодаря работе в оперативной памяти.
* Масштабируемость:
  + MongoDB и Redis лучше всего масштабируются горизонтально, что делает их подходящими для распределённых систем.
  + PostgreSQL и MySQL чаще масштабируются вертикально, но поддерживают кластеризацию для крупных систем.
* Потребление ресурсов:
  + Redis требует больших ресурсов оперативной памяти, так как хранит данные в ОЗУ.
  + MySQL и PostgreSQL более эффективны с точки зрения постоянного хранения данных, особенно при работе на серверах среднего уровня.
* Поддержка ACID:
  + PostgreSQL и MySQL предоставляют полную поддержку транзакций, что делает их надёжными для финансовых систем.
  + MongoDB поддерживает ACID только на уровне одной операции или транзакции в пределах одного документа.

**Числовое подтверждение**

* Производительность запросов:
  + PostgreSQL выполняет SELECT запросы за ~1.5 мс, что немного уступает MySQL (~1.2 мс), однако при увеличении сложности запросов PostgreSQL становится более эффективным благодаря поддержке сложных индексов.
  + MongoDB выполняет SELECT запросы с задержкой около ~1.8 мс, так как её основная специализация — работа с большими массивами данных, а не высокой скоростью транзакций.
  + Redis здесь лидер с минимальной задержкой (<0.1 мс), что объясняется хранением данных в оперативной памяти.
* Скорость записи:
  + MySQL показывает преимущество в скорости записи с ~1.8 мс против ~2.0 мс у PostgreSQL, что делает его более привлекательным для задач с интенсивными вставками данных.
  + MongoDB имеет задержку записи около ~2.5 мс, так как операции с JSON-документами требуют больше обработки. Redis вновь быстрее всех: INSERT операции практически мгновенны (<0.1 мс).
* Операции в секунду (RPS):
  + Redis достигает ~80,000 операций в секунду на типичных нагрузках, что значительно превышает возможности реляционных баз.
  + MySQL и PostgreSQL работают в пределах ~20,000–25,000 RPS, что вполне достаточно для большинства транзакционных систем.
  + MongoDB демонстрирует около ~18,000 RPS, что типично для NoSQL систем.
* Пиковая нагрузка:
  + Redis также выигрывает по максимальному числу операций (до 1,000,000 RPS), делая его идеальным выбором для кэширования или обработки очередей.
  + MySQL и PostgreSQL способны поддерживать до ~500,000–600,000 RPS на оптимизированных кластерах.
  + MongoDB уступает с пиком около ~400,000 RPS, но остаётся приемлемым выбором для систем с высокой параллельностью.
* Потребление ресурсов:
  + MySQL экономичен, потребляя ~150 МБ памяти на обычных нагрузках.
  + PostgreSQL использует больше ресурсов (~200 МБ), но оправдывает это расширенными функциями аналитики.
  + MongoDB требует ~300 МБ памяти, особенно при работе с крупными коллекциями JSON-документов.
  + Redis имеет гибкое потребление памяти, варьирующееся от ~100 МБ до нескольких гигабайт, в зависимости от объёма данных.

В результате проведённого анализа был сделан выбор в пользу PostgreSQL, поскольку эта система управления базами данных сочетает в себе высокую производительность и надёжность, что делает её оптимальным решением для сложных приложений. PostgreSQL демонстрирует стабильное время выполнения запросов даже при интенсивных нагрузках, а её архитектура обеспечивает эффективное выполнение сложных аналитических операций и транзакций.

Кроме того, поддержка расширенного SQL, разнообразных типов индексов и строгого соблюдения принципов ACID делает PostgreSQL предпочтительным выбором для систем, где требуется высокая надёжность, точность обработки данных и возможность масштабирования. Её универсальность позволяет адаптироваться под широкий спектр задач, включая аналитические вычисления, обработку больших объёмов данных и управление транзакционными процессами.

Остальные инструменты разработки

1. Docker vs Podman vs LXC

Docker — это популярный инструмент для контейнеризации приложений, однако есть и другие технологии, такие как Podman и LXC, которые также широко используются для контейнеризации.

Docker:

* Производительность: Docker использует контейнеризацию на уровне операционной системы с разделением ядер, что позволяет эффективно управлять ресурсами.
* Поддержка: Поддерживает множество операционных систем и платформ.
* Простота использования: Очень хорошо документирован и имеет широкий спектр инструментов для управления и развертывания контейнеров.
* Масштабируемость: Прекрасно масштабируется при использовании с Kubernetes.

Podman:

* Производительность: Подходит для сценариев, где безопасность имеет первостепенное значение, так как Podman не требует демона для работы.
* Безопасность: Podman работает без привилегий root-пользователя, что повышает безопасность при работе с контейнерами.
* Совместимость: Совместим с Docker API, что позволяет без проблем использовать уже готовые Docker-контейнеры.
* Масштабируемость: Требует дополнительных инструментов для масштабируемости, например, Kubernetes.

LXC:

* Производительность: LXC предоставляет более низкоуровневую контейнеризацию, чем Docker, и более похож на виртуальные машины.
* Гибкость: Более гибкий, позволяет настроить множество параметров системы внутри контейнера, что подходит для работы с виртуализированными приложениями.
* Масштабируемость: Требует более сложных настроек для работы в большом масштабе.

Docker — идеальный выбор для большинства случаев, особенно если требуется высокая скорость разработки и большая экосистема. Podman — отличная альтернатива с улучшенной безопасностью, а LXC подойдет для более специфичных случаев, где важна настройка и контроль на уровне ОС.

1. GitLab vs GitHub vs Bitbucket

GitLab, GitHub, и Bitbucket — это популярные сервисы для управления репозиториями Git, однако каждый из них имеет свои особенности.

GitLab:

* Производительность: GitLab предоставляет более полную экосистему для DevOps, включая CI/CD, управление проектами, мониторинг и даже возможности для облачного хостинга.
* Интеграция: GitLab интегрируется с широким спектром инструментов, и предоставляет собственную CI/CD систему.
* Гибкость: Предлагает возможность самохостинга, что идеально подходит для организаций, требующих полного контроля над своим кодом.
* Безопасность: GitLab имеет встроенные средства управления безопасностью, например, сканирование уязвимостей кода.

GitHub:

* Производительность: Предоставляет отличную платформу для хранения репозиториев, особенно для открытых проектов.
* Интеграция: GitHub интегрируется с огромным количеством сторонних инструментов и сервисов, что делает его гибким инструментом для различных рабочих процессов.
* Гибкость: Ориентирован больше на открытые проекты и менее на корпоративное использование, но для частных репозиториев предлагает отличный функционал.
* Безопасность: GitHub активно развивает свою безопасность, включая возможности для проверки зависимостей и приватные репозитории.

Bitbucket:

* Производительность: Bitbucket — это сильный инструмент для команд, работающих с продуктами Atlassian, такими как Jira и Confluence.
* Интеграция: Bitbucket тесно интегрируется с Jira и другими продуктами Atlassian, что делает его привлекательным для тех, кто уже использует эту экосистему.
* Гибкость: Лучше подходит для небольших и средних команд, хотя поддерживает как публичные, так и частные репозитории.
* Безопасность: Bitbucket предлагает хороший набор инструментов для обеспечения безопасности кодов и управляемых репозиториев.

GitLab — лучший выбор для DevOps-процессов с встраиваемыми функциями CI/CD, управления проектами и безопасности. GitHub подходит для открытых проектов и имеет обширную экосистему, а Bitbucket — идеален для тех, кто уже использует инструменты Atlassian.

1. Prometheus vs Zabbix vs Datadog

Prometheus — это мощный инструмент для сбора метрик и мониторинга, но есть и другие решения, такие как Zabbix и Datadog.

Prometheus:

* Производительность: Предназначен для мониторинга и хранения метрик в реальном времени, идеально подходит для микросервисных архитектур и работы с временными рядами.
* Гибкость: Использует язык запросов PromQL, что позволяет гибко извлекать данные и строить метрики.
* Интеграция: Отлично интегрируется с Kubernetes и другими контейнеризованными решениями.
* Масштабируемость: Prometheus масштабируется для работы с большими объемами данных, но требует настроек для работы с распределенными системами.

Zabbix:

* Производительность: Преимущественно ориентирован на мониторинг инфраструктуры и сетевых устройств, также поддерживает сбор метрик.
* Гибкость: Более статичен в настройке, но предоставляет широкие возможности для мониторинга серверов и сетевых компонентов.
* Интеграция: Хорошо работает с серверными системами и сетями, но не столь гибок при мониторинге микросервисов.
* Масштабируемость: Мощный инструмент для масштабируемых решений, но может требовать дополнительных ресурсов для управления большим количеством агентов.

Datadog:

* Производительность: Предоставляет облачный сервис мониторинга с продвинутыми возможностями анализа и визуализации.
* Гибкость: Служба ориентирована на автоматическое масштабирование и обладает высокой гибкостью в сборе метрик.
* Интеграция: Идеален для облачных приложений и сервисов с автоматической интеграцией в различные инфраструктуры.
* Масштабируемость: Может работать с любым количеством узлов и контейнеров, предоставляя точные метрики.

Prometheus лучший выбор для мониторинга в микросервисных приложениях благодаря своей гибкости и интеграции с Kubernetes. Zabbix будет лучше подходить для мониторинга серверных инфраструктур, а Datadog идеально подходит для облачных и высоконагруженных сервисов.

1. Nginx vs Apache vs Traefik

Nginx — это один из самых популярных веб-серверов и обратных прокси, но существуют и другие решения, такие как Apache и Traefik.

Nginx:

* Производительность: Отличается высокой производительностью благодаря своей асинхронной архитектуре.
* Гибкость: Подходит как для проксирования, так и для балансировки нагрузки и кэширования.
* Масштабируемость: Отлично масштабируется для обработки большого трафика и работы с контейнеризованными приложениями.
* Конфигурация: Имеет более простую и понятную конфигурацию по сравнению с Apache.

Apache:

* Производительность: Apache использует многопоточную модель, что может повлиять на производительность при высоких нагрузках.
* Гибкость: Предлагает множество конфигураций и модулей для расширения функциональности.
* Масштабируемость: Меньше подходит для распределенных систем и микросервисов по сравнению с Nginx.
* Конфигурация: Требует больше времени на настройку и управление по сравнению с Nginx.

Traefik:

* Производительность: Ориентирован на работу в контейнеризованных и микросервисных приложениях, с хорошей интеграцией с Docker и Kubernetes.
* Гибкость: Автоматически обнаруживает и настраивает маршруты для новых контейнеров.
* Масштабируемость: Отлично подходит для динамичных, микросервисных приложений.
* Конфигурация: Простота конфигурации, особенно для работы с контейнерами и в облаке.

Nginx — оптимальный выбор для большинства случаев, особенно для высокопроизводительных приложений и веб-сервисов. Traefik лучше для динамичных микросервисных приложений, а Apache подходит для старых приложений с большими требованиями к настройке.

Все инструменты, которые были выбраны, являются мощными и эффективными решениями для различных аспектов разработки и эксплуатации приложений. Они обеспечивают высокую производительность, безопасность и гибкость, а их аналоги, хотя и имеют свои преимущества, в целом не обеспечивают такую же степень интеграции и удобства, как выбранные вами решения.

Заключение

В рамках данного проекта была разработана и развернута высокотехнологичная информационная система, направленная на поддержку и модернизацию процессов в ракетно-космической отрасли. Проект охватывал полный цикл разработки, от анализа требований и проектирования архитектуры до развертывания инфраструктуры, интеграции инструментов автоматизации и мониторинга. Особенностью данного подхода стало проведение сравнительного анализа технологий и инструментов, доступных на рынке, что позволило выбрать оптимальные решения, максимально соответствующие требованиям проекта.

Архитектурные решения и технологический выбор

Основой системы стала микросервисная архитектура, которая продемонстрировала свои преимущества в масштабируемости, гибкости и устойчивости к изменениям. Контейнеризация с использованием Docker и оркестрация Kubernetes упростили управление компонентами и их адаптацию к изменяющимся бизнес-потребностям. Выбор PostgreSQL в качестве базы данных был обоснован её превосходной производительностью, стабильностью при сложных аналитических запросах и высокой надежностью.

При выборе технологий для разработки основного функционала был проведён детальный сравнительный анализ языков программирования, таких как Python, Java, Node.js и Ruby, что позволило обоснованно выбрать Go (Golang) благодаря его скорости, эффективному управлению памятью, безопасности и поддержке параллелизма. Этот выбор значительно повысил эффективность реализации микросервисной архитектуры и производительности системы в целом.

Инфраструктура и автоматизация процессов

Серверная инфраструктура была развернута с использованием Nginx в роли веб-сервера и обратного прокси, а также Redis для кэширования и PostgreSQL для работы с данными. Особое внимание было уделено автоматизации процессов разработки и развертывания. GitLab CI/CD был выбран в качестве инструмента для реализации непрерывной интеграции и доставки, что позволило минимизировать время вывода новых функций на рынок и обеспечить стабильность выпускаемых обновлений.

Проведённое сравнение с другими платформами для автоматизации, такими как GitHub Actions и Jenkins, подтвердило, что GitLab предоставляет наиболее подходящие возможности для интеграции с выбранной технологической инфраструктурой и оптимизации рабочего процесса.

Мониторинг, логирование и аналитика

Для обеспечения стабильной работы системы была реализована комплексная система мониторинга и логирования. Prometheus и Grafana позволили создать централизованную платформу для сбора и визуализации метрик, что обеспечило оперативный анализ производительности системы. Использование ELK Stack дополнило мониторинг возможностями глубокого анализа логов, а Alertmanager был интегрирован для обеспечения автоматического оповещения команды о возникновении проблем. Эти инструменты были выбраны на основе анализа их альтернатив, таких как Zabbix, Datadog и Splunk, с учётом критериев масштабируемости, интеграции и удобства использования.

Практическая значимость и перспективы

Созданная система соответствует современным стандартам надёжности, безопасности и производительности. Она предоставляет платформу, которая может эффективно адаптироваться к меняющимся требованиям отрасли. Возможности её дальнейшего развития включают в себя расширение автоматизации, использование технологий искусственного интеллекта для анализа данных, а также интеграцию с внешними системами.

Сравнительный анализ инструментов и технологий на рынке стал важной частью работы, позволившей выбрать оптимальные решения для реализации системы. Этот подход обеспечил высокую эффективность работы, снизил издержки и ускорил процесс разработки.

Результатом стала современная информационная система, способная поддерживать и развивать процессы в ракетно-космической отрасли. Она обеспечивает высокий уровень надёжности, производительности и гибкости, создавая долгосрочные перспективы для повышения конкурентоспособности отрасли и внедрения инновационных решений.

Дальнейшие направления развития:

Углубление автоматизации тестирования и развертывания для ещё большего сокращения времени вывода новых функций в эксплуатацию.

Расширение возможностей анализа данных и применения технологий искусственного интеллекта для оптимизации процессов.

Подключение дополнительных сервисов для интеграции с внешними системами и решения более сложных задач.

Таким образом, проект создал надежную основу для поддержки и развития высокотехнологичных процессов в ракетно-космической отрасли, обеспечив долгосрочную перспективу для повышения её эффективности и конкурентоспособности.

Список литературы

1. Калашников А.Д., Ткачева Д.А, Жидков Е.И., Вдовкина Е.Д., Горячкин Ь.С., Терехов В.И. [Многофункциональная интеллектуальная информационная система эволюции развития и современного состояния ракетно-космической отрасли](https://www.elibrary.ru/item.asp?id=54327004&pff=1). // XLVII Академические чтения по космонавтике 2023: сборник тезисов докладов - М., 2023 - С. 32 - 34.
2. Горячкин Б.С., Кузнецов М.И., Майорова В.И., Терехов В.И. [Интеллектуальный информационный комплекс «Ракетно-космические адреса России»: концепция проекта](https://www.elibrary.ru/item.asp?id=54326829&pff=1) // XLVII Академические чтения по космонавтике 2023: сборник тезисов докладов - М., 2023 - С. 13 – 15.
3. Кузнецов М.И. Ракетно-космические адреса Московского региона. Цандер // XLIV академические чтения по космонавтике (Королевские чтения – 2020): сб. тез.: в 4 т. М.: Изд-во МГТУ им. Н.Э. Баумана, 2020 Т. 1 с. 19-21
4. Кузнецов М.И. «Ракетно-космические» адреса взаимодействия Цандера, Королёва, Победоносцева в окрестностях МВТУ – МММИ // XLVI Академические чтения по космонавтике (Королёвские чтения — 2022): сб. тез.: в 4 т. М.: Изд-во МГТУ им. Н.Э. Баумана, 2022. Т. 1. С. 17–19.
5. Кузнецов М.И. Космическая дорога наукоградов Московского региона // XL Академические чтения по космонавтике (Королёвские чтения — 2016): сб.тез.:М.:, Изд-во МГТУ им. Н.Э. Баумана, 2016. С. 298
6. ГОСТ 34.601-90. Автоматизированные системы. Стадии создания. — Введённый государственным стандартом процесс создания автоматизированных систем, описывающий требования к этапам проектирования.
7. ГОСТ Р 19.105-2020. Единая система программной документации. Общие требования к программной документации. — Стандарты на оформление технической документации в разработке программного обеспечения.
8. ISO/IEC 25010:2011. Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE). — Стандарты качества, применяемые к системам и программному обеспечению, в том числе для оценки функциональных характеристик и надёжности.
9. Kubernetes Documentation. Kubernetes.io — https://kubernetes.io/docs/ — Официальная документация по Kubernetes, охватывающая архитектурные аспекты, развертывание и управление кластером.
10. Docker Documentation. Docker.com — https://docs.docker.com/ — Руководство по настройке и управлению контейнерами, включая инструменты оркестрации и CI/CD.
11. PostgreSQL Documentation. PostgreSQL.org — https://www.postgresql.org/docs/ — Документация по PostgreSQL, охватывающая вопросы проектирования, администрирования и оптимизации баз данных.
12. Prometheus Documentation. Prometheus.io — https://prometheus.io/docs/ — Инструменты мониторинга и их настройка для сбора и визуализации метрик.
13. Grafana Documentation. Grafana.com — https://grafana.com/docs/ — Руководство по визуализации данных и созданию дашбордов.