УДК 004.75, 004.514, 004.432, 004.8

ПРОЕКТИРОВАНИЕ И ПРОТОТИПИРОВАНИЕ ИНТЕЛЛЕКТУАЛЬНОЙ ОБЛАЧНОЙ СРЕДЫ УПРАВЛЕНИЯ СЛОЖНЫМИ ПРОГРАММНЫМИ ПРОДУКТАМИ

Бурмич Даниил Евгеньевич

Разработчик;
ООО «МВС»;
Россия, 115432, г. Москва, пр-кт Андропова, 18;
e-mail: nagl.dk8@gmail.com.

В работе рассматривается проектирование и прототипирование интеллектуальной облачной среды управления сложными программными продуктами. Работа состоит в проверке архитектурного подхода, который объединяет среду командной разработки, репозитории, CI/CD-контур и интеллектуальный компонент. В ходе системного анализа выявляются проблемы организации разработки, формируются требования к среде, выполняется обзор классов существующих решений и сопоставляются варианты реализации. Описывается архитектура прототипа на базе Space On-Premises, Docker Compose и внешних модулей, а также интеграция интеллектуального компонента с внешними и локально развернутыми моделями генеративного искусственного интеллекта.

Ключевые слова: интеллектуальная облачная среда, DevOps, CI/CD, Space On-Premises, генеративный искусственный интеллект, LLM, чат-бот, Docker, прототипирование.

Для цитирования:

Бурмич Д. Е. Проектирование и прототипирование интеллектуальной облачной среды управления сложными программными продуктами // Системный анализ в науке и образовании. 2026. № 3. С. 1-12. EDN: BDYQQE. URL: https://sanse.ru/index.php/sanse/article/view/731.

DESIGN AND PROTOTYPING OF AN INTELLIGENT CLOUD ENVIRONMENT FOR MANAGING COMPLEX SOFTWARE PRODUCTS

Burmich Daniil E.

Developer;
LLC "MWS";
18 Andropova Pr-t., Moscow, 115432, Russia;
e-mail: nagl.dk8@gmail.com.

The paper considers the design and prototyping of an intelligent cloud environment for managing complex software products. The aim is to validate an architectural approach that combines a collaborative development environment, repositories, a CI/CD circuit and an intelligent component. Based on systems analysis, the paper identifies problems in software development organization, formulates requirements, reviews classes of existing solutions and compares implementation options. The paper describes the prototype architecture based on Space On-Premises, Docker Compose and external modules, as well as intelligent component integration with external and locally deployed generative AI models.

Keywords: intelligent cloud environment, DevOps, CI/CD, Space On-Premises, generative artificial intelligence, LLM, chatbot, Docker, prototyping.

For citation:

Burmich D. E. Design and prototyping of an intelligent cloud environment for managing complex software products. *System analysis in science and education,* 2026;(3):1-12 (in Russ). EDN: BDYQQE. Available from: https://sanse.ru/index.php/sanse/article/view/731.

Статья находится в открытом доступе и распространяется в соответствии с лицензией Creative Commons «Attribution» («Атрибуция») 4.0 Всемирная (CC BY 4.0) https://creativecommons.org/licenses/by/4.0/deed.ru

Введение

Создание программных продуктов высокой сложности требует согласованной работы нескольких контуров: управления задачами и командами, хранения проектной документации, контроля версий, проверки кода, сборки, тестирования, развертывания и сопровождения. Когда эти процессы распределены между разными инструментами, рабочий контекст фрагментируется: участники проекта переходят между сервисами, вручную дублируют сведения, а новые сотрудники дольше входят в предметную область.

Эта постановка близка к логике *DevOps* и *continuous delivery*: скорость и устойчивость поставки изменений зависят не только от инструментов сборки, но и от согласованности разработки, тестирования, эксплуатации, коммуникации и обратной связи [10-12]. В настоящей работе облачная среда понимается как организационно-технический контур, в котором рядом находятся рабочие артефакты, события разработки и средства автоматизации.

Поэтому разработка облачной среды в рамках этой работы рассматривается как инженерный процесс, который от выявления организационных проблем ведет шаг за шагом к набору проверяемых технических решений. В этом процессе важен не только выбор платформы; последовательно уточняются требования, фиксируются ограничения, строится архитектура, проверяется развертывание и добавляется интеллектуальный помощник в рабочий контур.

Цель работы в том, чтобы показать процесс создания интеллектуальной облачной среды управления сложными программными продуктами от анализа исходной ситуации до получения прототипа, в котором единое пространство разработки совмещено с автоматизацией и чат-ботом, поддерживающим обращение к внешним и локально развернутым моделям генеративного искусственного интеллекта.

Научно-практический результат работы состоит в архитектурном подходе к объединению среды командной разработки, *CI/CD*-контура и интеллектуального чат-бота в единую управляемую систему. Отличительная особенность подхода заключается в подключении внешних и локально развернутых моделей генеративного искусственного интеллекта через адаптеры. За счет этого используемая модель заменяется без изменения пользовательского сценария работы со средой.

1. Выявление проблем и формирование требований

Первый этап проектирования связан с определением проблем, которые возникают при разработке сложных программно-технологических решений. Наиболее заметными оказываются потери времени на координацию, поиск актуальной информации, повторяющиеся операции сборки и тестирования, а также необходимость быстро передавать новым участникам проекта накопленный контекст.

Для перехода от общей постановки проблемы к проектным требованиям рабочий процесс разделяется на несколько функциональных контуров. В отдельные группы выделяются управление задачами и ролями, взаимодействие команды, репозитории и *code review*, документация, автоматизация сборки и развертывания, а также интеллектуальная поддержка пользователя. Такой разбор позволяет рассматривать будущую среду как систему взаимосвязанных подсистем, а не как набор разрозненных сервисов.

На основе анализа формируются требования к среде:

1. поддержка локального или контролируемого серверного развертывания;
2. ведение репозиториев, проверка изменений и работа с версиями;
3. управление задачами, командами, ролями и правами доступа;
4. единое хранение проектной документации, файлов и обсуждений;
5. автоматизация сборки, тестирования и развертывания программных компонентов;
6. расширение через *API, SDK, webhooks* или собственные приложения;

7. подключение интеллектуального компонента, способного отвечать на вопросы пользователя и использовать проектный контекст.

Последний пункт требований задает границу выбранной концепции. Интеллектуальный компонент не заменяет разработчика или систему управления проектом. Его задача состоит в снижении порога входа в проект, ускорении поиска информации и поддержке типовых вопросов на естественном языке внутри той же среды, где ведется работа команды.

2. Обзор существующих решений и выбор технологической основы

После формулирования требований сопоставляются два пути реализации: создание собственной платформы с нуля и построение среды на базе готового корпоративного решения с последующим расширением. Первый путь дает большую свободу проектирования, однако требует разработки большого числа базовых модулей: репозиториев, средств коммуникации, автоматизации, управления задачами и модели прав. Второй путь быстрее переводит работу от анализа к проверке рабочего процесса, поэтому каждый аналитический шаг направлен на уточнение того, какие риски закрываются готовой платформой, а какие компоненты необходимо реализовать отдельно.

Чтобы выбор технологической основы не сводился к подстановке конкретного продукта, рассматриваются несколько классов решений: системы контроля версий и *code review*, *CI/CD*-платформы, трекеры задач, базы знаний, корпоративные коммуникационные среды и интегрированные платформы командной разработки. Отдельные инструменты, такие как *Jenkins*, *TeamCity* или *Circle CI*, сильны в автоматизации сборки и тестирования, но не закрывают полный контур управления задачами, коммуникациями и проектным контекстом. Системы класса *Jira*, *YouTrack* или *Confluence*, напротив, помогают организовать задачи и знания, но требуют отдельной интеграции с репозиториями и *CI/CD*.

В качестве целевого направления выбирается не разработка всех подсистем с нуля, а построение прототипа на базе готового корпоративного ядра с расширением через *API, SDK, webhooks* и внешние сервисы. Такой вариант дает возможность проверить архитектурную связность решения до принятия решения о промышленной реализации или переносе архитектуры на другой набор инструментов. Таблица 1 содержит сводные данные *SWOT*-анализа, используемые для выбора направления реализации.

Табл. 1. Сводные данные SWOT-анализа концепции облачной среды

Блок анализа	**Данные анализа**	**Проектный вывод**
Сильные стороны	быстрое развертывание платформы на базе контейнеров; собственное размещение на сервере; контроль над инфраструктурой	среду целесообразно проверять на локальном стенде, не привязываясь к внешнему публичному облаку
Слабые стороны	ограниченный доступ при локальном размещении; отсутствие мобильного сценария без внешнего адреса; сложность дальнейшей настройки	требуются механизм внешнего доступа, документированная конфигурация и возможность поэтапного расширения
Возможности	востребованность изолированных сервисов у целевых пользователей; возможность подключения дополнительных компонентов и заказчиков	ядро системы должно поддерживать интеграции, роли пользователей и развитие сервисов без полной переработки архитектуры
Угрозы	сложность автоматизации платформы; риск отказа локального сервера; необходимость обучения специалистов	решение должно опираться на понятные сценарии автоматизации, контейнерное развертывание и резервируемую серверную инфраструктуру

Для уточнения рисков выполняется причинно-следственная проверка. Ее результат показывает, какие причины влияют на разработку и какие проектные решения закладываются для их компенсации (см. табл. 2).

Табл. 2. Причинно-следственная проверка рисков проектирования

Группа причин	**Выявленная причина**	**Реакция в проектировании**
Целевая компания	возможный разрыв сотрудничества; отсутствие формализованного технического задания	требования фиксируются через проверяемые функции: коммуникации, репозитории, задачи, документация, автоматизация и расширяемость
Окружение	нехватка времени и недостаточная полнота практических материалов по интеграции	вместо разработки собственного ядра выбирается путь сборки среды из готовых сервисов и последовательных проверочных сценариев
Оборудование	сложное внутреннее устройство сервера; риск недостаточной производительности при локальном размещении	развертывание выносится в контейнерную конфигурацию, чтобы отделить сервисы среды от настроек хоста и упростить повторный запуск
Человек	нехватка знаний в области автоматизации; отсутствие компетенций по отдельным компонентам	предпочтение отдается платформе с готовыми сценариями командной работы, поддержкой скриптов автоматизации и *API* для расширений

После фиксации рисков выполняется выбор технологической основы. В качестве альтернатив рассматриваются *Space*, *Jenkins*, *Buddy*, *TeamCity* и *Circle CI*. Сравнение проводится не только по возможностям непрерывной интеграции, а по тому, может ли платформа стать ядром общей рабочей среды. Наибольший вес получают социальные элементы, интеграционный *API* и поддержка *Git*: эти критерии напрямую связаны с требованием объединить работу команды, репозитории, автоматизацию и внешние сервисы. Веса критериев определяются экспертно на основе парного сопоставления значимости требований для целевой архитектуры. Таблица 3 содержит итоговые веса критериев выбора.

Табл.3. Веса критериев при выборе технологической основы

Критерий	**Вес**	**Значение критерия для проекта**
Социальные элементы среды	0,3457	наличие коммуникаций, профилей, проектных пространств и механизмов командного взаимодействия
Интеграционный *API*	0,2568	возможность подключать внешние сервисы, чат-бота, интеллектуальный компонент и обработчики событий
Поддержка *Git*	0,1606	ведение кода, ревью изменений и связь репозитория со сценариями автоматизации
Развертывание на собственном сервере	0,1450	контроль инфраструктуры, изоляция данных и возможность проверки среды на локальном сервере
Удобство интерфейса	0,0918	снижение порога входа для пользователей и упрощение ежедневных операций внутри среды

Полученные веса меняют логику выбора: узкие *CI/CD*-системы закрывают часть задач автоматизации, но не дают полноценного рабочего пространства для команды. *Space*, напротив, совмещает репозитории, *code review*, задачи, чаты, пакеты, автоматизацию и средства расширения через *HTTP API*, *webhooks* и *SDK* [1]. Поэтому платформа выбирается как ядро прототипа, а отсутствующие подсистемы рассматриваются как внешние интеграционные модули.

При интерпретации результата учитывается жизненный цикл используемой платформы. *JetBrains* объявила о прекращении развития *Space* в исходном виде и о переносе акцента на решения

для *Git*-хостинга и *code review* [8]. Для *Space On-Premises* производитель допускает дальнейшее использование уже развернутых экземпляров при выполнении условий лицензирования [9]. Поэтому в настоящей работе *Space* рассматривается как экспериментальная платформа для проверки архитектурного подхода, а не как универсальная рекомендация для новых промышленных внедрений. Содержательный результат работы переносим на другие связки *Git*-хранилища, трекера задач, *CI/CD* и коммуникационного инструмента, если они предоставляют *API* и механизмы событийной интеграции. Рис. 1 показывает, какие компоненты исходной архитектурной схемы закрываются платформой *Space*.

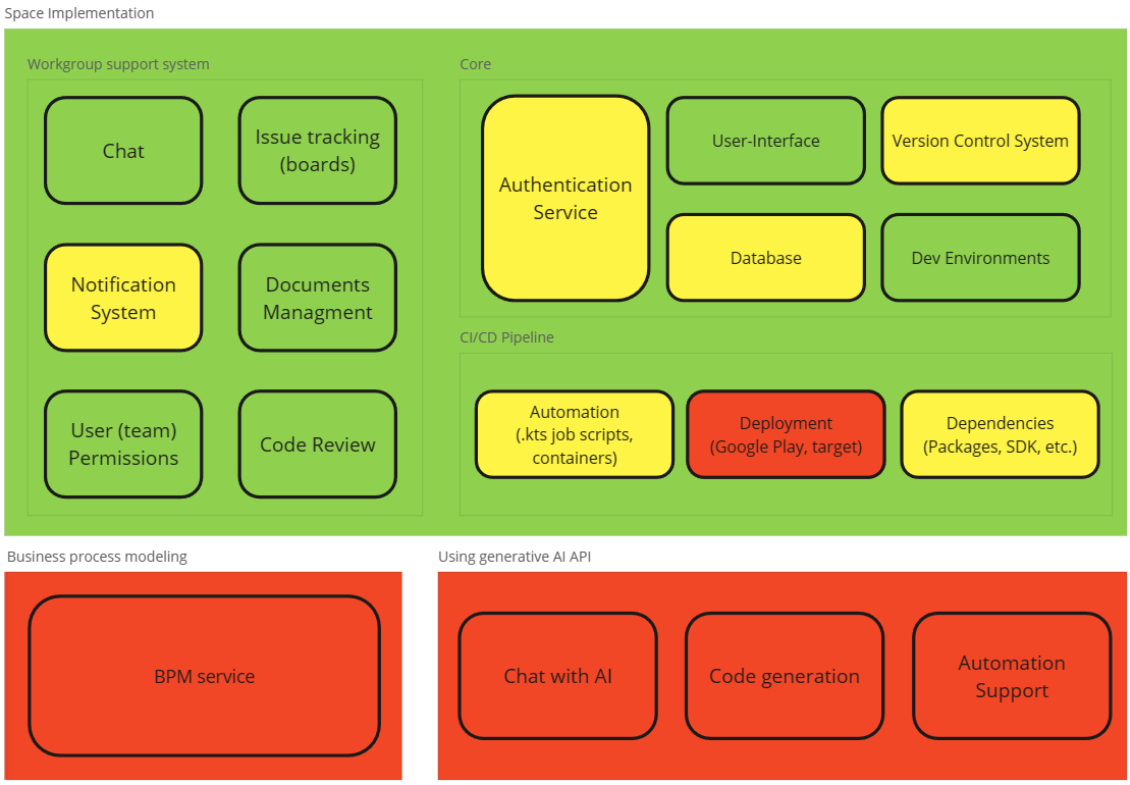


Рис. 1. Схема архитектуры облачной среды с выделением компонентов, закрываемых платформой Space

Отдельно рассматривается способ развертывания. Для промышленного масштабирования подходит *Kubernetes*, однако для проверки работоспособности архитектуры и настройки первичного прототипа достаточен *Docker Compose*, так как он позволяет описать несколько связанных контейнеров в одной конфигурации и ускоряет цикл запуска, проверки и корректировки сервисов [2]. Поэтому на этапе реализации прототипа выбирается *Docker Compose*, а переход к более сложной оркестрации остается направлением дальнейшего развития.

В итоге *SWOT*-матрица задает границы решения, причинно-следственная проверка показывает основные риски реализации, а взвешивание критериев объясняет выбор ядра платформы. Дальнейшее проектирование строится от этого результата, где *Space* принимается как центральный контур совместной работы, *Docker Compose* используется как средство быстрого прототипирования, а интеллектуальные и процессные функции задаются как расширяемые компоненты.

3. Проектирование состава и архитектуры среды

На основе результатов анализа формируется архитектурная схема среды (см. рис. 2). Проектирование идет от состава компонентов: какие функции закрывает платформа *Space*, какие сервисы нужны для ее устойчивой работы, какие подсистемы добавляются отдельно и через какие интерфейсы они взаимодействуют.

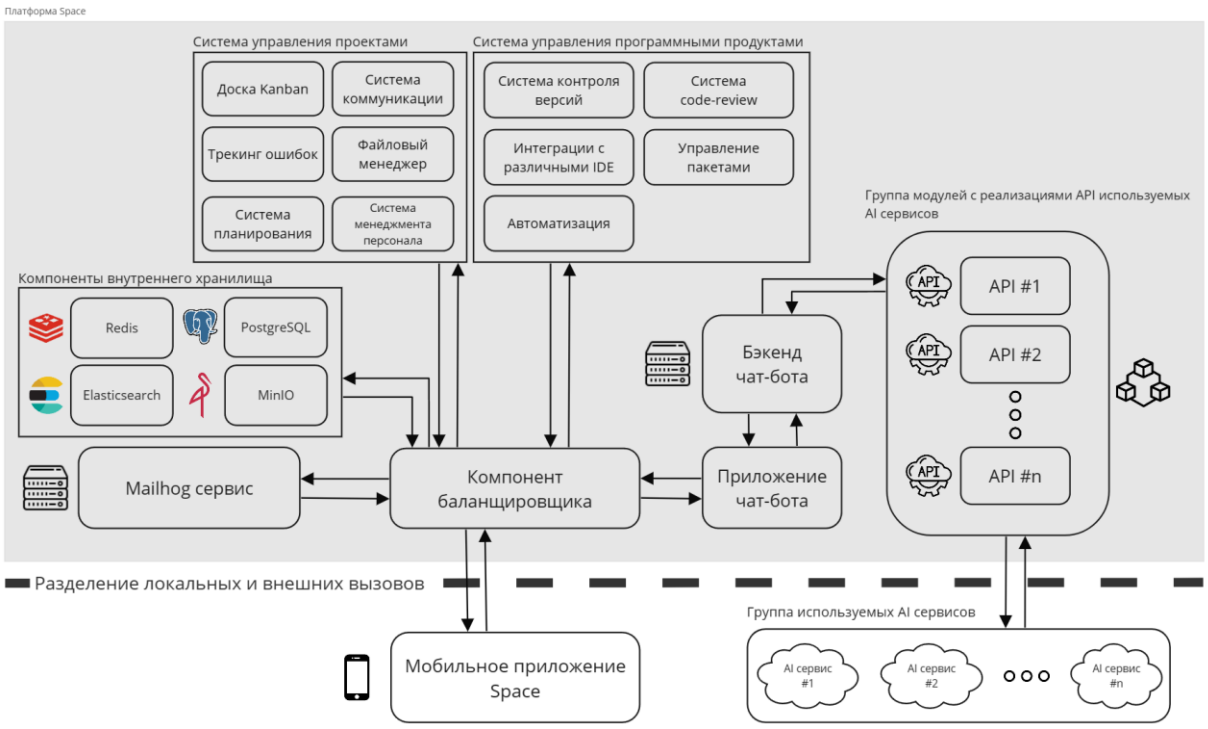


Рис. 2. Архитектура интеллектуальной облачной среды

В центре архитектуры находится *Space* как ядро корпоративного пространства. Вокруг него формируются подсистемы управления проектами, репозиториями, *code review*, коммуникациями, планированием, хранением данных и автоматизацией. Для внутренней работы платформы используются *Redis, PostgreSQL, Elasticsearch, MinIO* и *MailHog*. Такой состав позволяет разделить прикладные функции среды и сервисы хранения, индексации, файлового обмена и уведомлений.

Внешний контур архитектуры строится через балансировщик и *HTTP*-сервер. Такой вариант отделяет внутренние вызовы от обращений извне, централизует маршрутизацию и готовит основу для последующего усиления отказоустойчивости. В качестве *HTTP*-сервера может применяться *Nginx*, поскольку он поддерживает работу в роли *reverse proxy* и подходит для обработки входящих запросов к внутренним сервисам [3].

Интеллектуальный компонент проектируется как отдельная подсистема, а не как жестко встроенная часть платформы. Это важно, потому что в разных сценариях применяются разные модели: внешние сервисы удобны для быстрых экспериментов, а локальные модели нужны там, где требуется контролировать данные и инфраструктуру. Поэтому в архитектуру включается чат-бот, серверная часть которого принимает команды из *Space* и передает запросы в один из адаптеров *AI*-сервисов.

Разделение на приложение чат-бота, серверную логику и адаптеры моделей сохраняет расширяемость. При появлении новой модели или сервиса достаточно добавить новый адаптер, не меняя общий сценарий взаимодействия пользователя со средой. Благодаря этому интеллектуальный компонент становится частью рабочей среды, а не внешним справочным инструментом. Рисунок 3 показывает схему взаимодействия чат-бота с сервисами генеративного искусственного интеллекта.

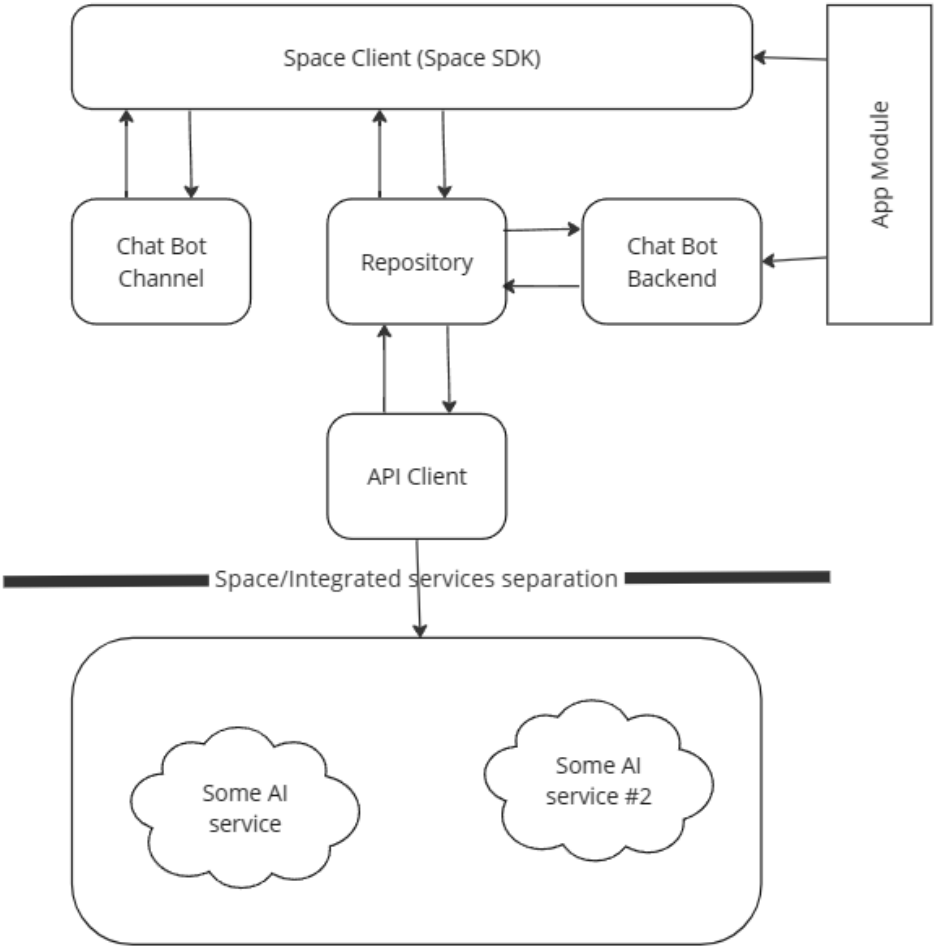


Рис. 3. Схема чат-бота с интеграцией сервисов генеративного искусственного интеллекта

4. Рабочий процесс реализации прототипа

Реализация прототипа начинается с подготовки тестовой инфраструктуры. На виртуальной машине разворачивается серверная среда, устанавливаются компоненты *Docker*, создается рабочий каталог и подготавливается файл *docker-compose.yml*. Через него запускаются контейнеры платформы и связанные сервисы, что дает возможность воспроизводимо проверять изменения конфигурации и быстро возвращаться к рабочему состоянию стенда [2].

После первичного запуска выполняются настройки, без которых среда не используется как единый рабочий контур: задается внешний адрес, настраиваются параметры уведомлений, добавляется *MailHog* для проверки почтовых сообщений, подготавливается *nginx* для внешнего доступа и маршрутизации, а также создаются базовые сущности платформы: пользователи, команды, проекты, репозитории и доски задач.

Настройка идет итерационно. Сначала проверяется запуск контейнеров и доступность внутренних сервисов. Затем устраняются проблемы с адресами переходов, почтовыми уведомлениями и доступом к среде из внешней сети. После этого создается минимальная проектная структура с командой, тестовым проектом, репозиториями, задачами и демонстрационными артефактами. Такой порядок отделяет инфраструктурные ошибки от ошибок настройки платформы.

Следующий шаг состоит в проверке автоматизации. В *Space* сценарии сборки и развертывания описываются в файле *.space.kts*, который хранится в репозитории проекта и позволяет запускать задачи на воркерах платформы [1]. В прототипе создаются задачи для сборки тестового приложения, проверки выполнения скриптов и отображения результатов во вкладке автоматизации.

При настройке автоматизации проверяется не только сам факт запуска скрипта, но и полный рабочий цикл появления задания во вкладке автоматизации, подключение воркера, выполнение команд сборки, возврат статуса и доступность журнала выполнения. В качестве тестового сценария используется сборка проекта через *Gradle*-контейнер. Такой сценарий показывает, что автоматизация связана с репозиторием и может входить в регулярный процесс разработки.

Рабочий процесс реализации показывает, что среда *Space* связывает код, задачи и автоматизацию в одном цикле. Изменение в репозитории сопровождается сборкой, результат выполнения отображается рядом с проектными артефактами, а команда получает единое место для контроля состояния работ. Этот этап подтверждает, что выбранная платформа подходит не только как хранилище кода, но и как контур управления инженерным процессом. Рисунок 4 показывает пример вкладки со сценариями автоматизации.



Рис. 4. Вкладка скриптов автоматизации в Space

5. Реализация интеллектуального компонента

После проверки базовой среды реализуется интеллектуальный компонент. Сначала в *Space* создается приложение чат-бота, которому назначаются необходимые разрешения. Серверная часть принимает сообщения пользователей, обрабатывает команды и возвращает ответ в канал. На раннем этапе это позволяет проверить регистрацию приложения, выдачу прав доступа и корректность обмена сообщениями через *SDK* платформы [1].

Логика чат-бота разделяется на несколько уровней. Первый уровень отвечает за прием событий из *Space* и распознавание команд пользователя. Второй уровень формирует запрос к заданной модели и приводит ответ к единому формату. Третий уровень отправляет результат обратно в канал. Такое модульное разделение важно для дальнейшего развития, так как изменение модели или способа обращения к ней не требует переработки всей логики взаимодействия с платформой [7].

Для тестирования входящих запросов локальный сервер чат-бота выводится во внешний доступ. Для этого применяется *ngrok*, который создает защищенный туннель к локальному сервису и выдает публичный адрес для внешних *callback*-запросов [4]. Такой режим ускоряет отладку, а серверная часть изменяется локально, без постоянного развертывания на отдельной инфраструктуре.

Затем добавляется команда *prompt*, через которую пользователь отправляет текстовый запрос. Внешний сценарий обработки реализуется через *Replicate*, предоставляющий *API* для запуска моделей генеративного искусственного интеллекта [5]. Для локального сценария готовится интеграция с *Ollama*, чтобы с ее помощью открытые модели запускались в собственной локальной инфраструктуре [6]. В интерфейсе чат-бота пользователь выбирает, какой тип модели использовать для ответа (см. рис. 5).

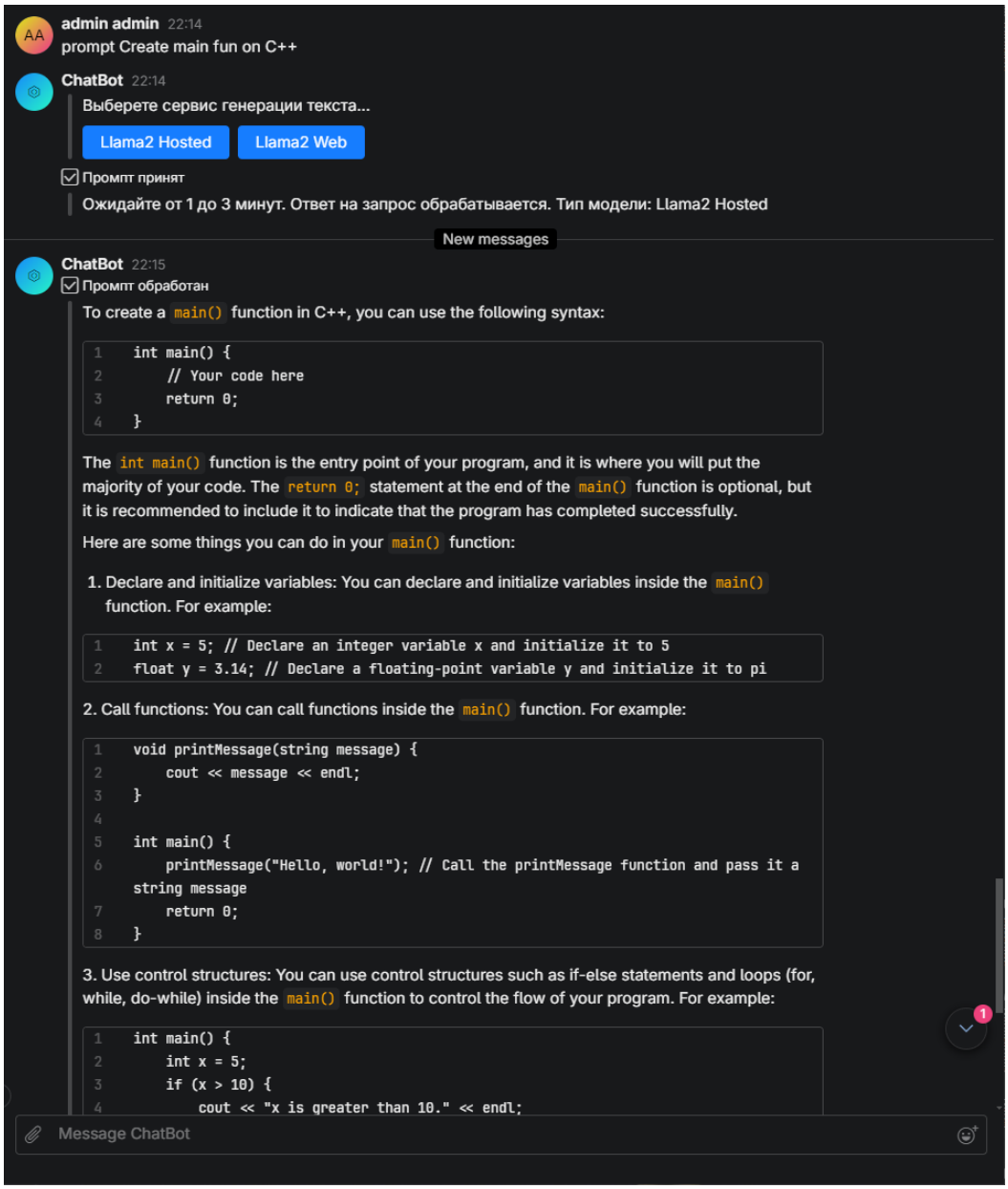


Рис. 5. Использование функционала с выбором модели ИИ

В ходе реализации отдельно проверяются особенности работы с длительными ответами локальных моделей. Генерация может занимать больше времени, чем обычный ответ серверного приложения, поэтому для серверной части увеличиваются тайм-ауты ожидания и добавляется промежуточное сообщение о том, что запрос принят в обработку.

Сравнение двух сценариев показывает разные области применения. Внешняя модель удобна для быстрой проверки гипотез и общих вопросов, но зависит от сетевого доступа, стоимости и политики передачи данных. Локальная модель лучше подходит для работы с внутренними материалами проекта, однако требует вычислительных ресурсов и отдельного сопровождения. Поэтому архитектура не фиксирует единственный вариант, а поддерживает смену адаптера в зависимости от требований к конкретному проекту.

Отдельное направление развития связано с подготовкой данных для настройки моделей на проектном контексте. Источниками могут служить документация, история задач, результаты *code review*, описания бизнес-процессов и типовые сценарии разработки. *Fine-tuning* не входит в обязательный состав прототипа, но может превратить интеллектуальный компонент из общего чат-интерфейса в помощника, работающего с внутренним контекстом проекта (см. рис. 6). Для практического внедрения также применим подход *retrieval-augmented generation*, при котором ответ формируется с учетом найденных фрагментов проектной документации без обязательного дообучения базовой модели [13].

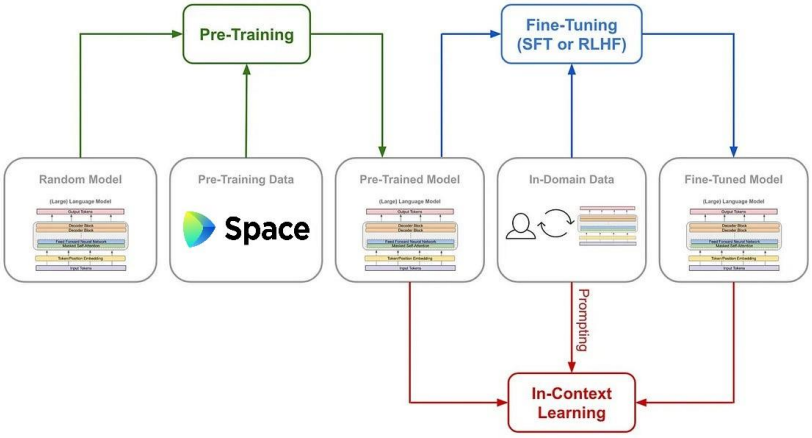


Рис. 6. Схема использования Fine-tuning при обучении модели на собственных данных

6. Результаты анализа и реализации

В результате формируется прототип интеллектуальной облачной среды, в котором проверяются основные этапы создания решения: постановка требований, выбор технологической основы, проектирование архитектуры, развертывание сервисов, настройка автоматизации, разработка и подключение чат-бота. Результат важен не только как набор настроенных компонентов, но и как подтверждение связности архитектурного подхода, в котором аналитические выводы последовательно переводятся в требования, на их основе строится архитектура, а затем архитектура проверяется на сценариях работы стенда.

Работоспособность прототипа оценивается по набору функциональных сценариев. Проверяются создание проекта и репозитория, добавление участников и задач, запуск автоматизации, просмотр результатов выполнения, отправка команды чат-боту, выбор внешней или локальной модели и получение ответа в рабочем канале (см. табл. 4).

Табл. 4. Проверка работоспособности прототипа по функциональным сценариям

Проверяемый сценарий	Что проверяется	Результат
Развертывание среды	Запуск контейнеров и доступность интерфейса платформы	Подтверждается
Создание проектного пространства	Пользователи, команды, проект, репозиторий и доска задач	Подтверждается
Автоматизация *CI/CD*	Запуск сценариев *CI/CD*, подключение воркера и журнал выполнения	Подтверждается
Интеграция внешней модели ИИ	Отправка запроса через чат-бот к внешнему *API*	Подтверждается
Интеграция локальной модели ИИ	Отправка запроса через чат-бот к локально развернутой модели	Подтверждается
Длительный ответ модели	Тайм-ауты ожидания и промежуточное уведомление пользователя	Подтверждается

Практическая ценность прототипа состоит в объединении нескольких контуров управления сложным программным продуктом. Команда получает доступ к задачам, репозиториям, коммуникациям, документации, автоматизации и интеллектуальной поддержке в единой среде. За счет этого уменьшается необходимость переносить контекст между разными инструментами, а типовые вопросы по проекту обрабатываются внутри рабочего пространства.

При этом у прототипа остаются ограничения. Тестовое развертывание служит для проверки архитектурной реализуемости, а не для оценки промышленной нагрузки. Для внедрения в промышленный контур требуется дополнительно проработать мониторинг, резервное копирование, хранение секретов, политики безопасности, процедуры обновления и регламент эксплуатации. Кроме

того, прекращение развития *Space* снижает применимость решения для новых внедрений, поэтому основной результат статьи следует рассматривать как переносимый архитектурный подход, а не как рекомендацию использовать именно эту платформу.

Наиболее перспективным направлением развития остается более глубокая связь интеллектуального компонента с проектными событиями и источниками данных. Чат-бот может не только отвечать на вопросы, но и анализировать ошибки сборки, предлагать шаги диагностики, формировать задачи, ссылаться на релевантные документы и помогать новым участникам проекта быстрее понимать текущий контекст разработки.

Заключение

Работа показывает процесс инжиниринга интеллектуальной облачной среды управления сложными программными продуктами. В ней рассматриваются этапы выявления проблем, формализации требований, обзора существующих решений, выбора технологической основы, проектирования архитектуры, настройки прототипа и интеграции интеллектуального компонента.

Анализ показывает, что использование готовой платформы корпоративной разработки как ядра прототипа сокращает объем базовой инфраструктурной разработки и переносит внимание на интеграцию, автоматизацию и интеллектуальную поддержку пользователей. Реализация стенда подтверждает возможность объединить репозитории, задачи, коммуникации, автоматизацию и чат-бота в едином рабочем контуре.

Дальнейшее развитие решения целесообразно направить на повышение надежности инфраструктуры, формализацию политик безопасности, расширение сценариев автоматизации и более глубокую интеграцию моделей генеративного искусственного интеллекта с проектными данными и событиями разработки. Предложенный подход удобно рассматривать как архитектурный шаблон, переносимый на другие платформы командной разработки с открытыми *API* и событийными механизмами интеграции.

Список источников

1. JetBrains Space : [technological platform] / JetBrains. – JetBrains s.r.o., 2000-2026. – URL: https://www.jetbrains.com/help/space/.
2. Docker Docs // Docker : [software platform]. – Docker Inc., 2013-2026. – URL: https://docs.docker.com/.
3. nginx documentation // Nginx: [web server]. – 2026. – URL: https://nginx.org/ru/docs/.
4. ngrok Docs // Ngrok : [cross-platform application]. – 2026. – URL: https://ngrok.com/docs/.
5. Documentation // Replicate: [cloud platform]. – URL: https://replicate.com/docs.
6. Ollama: [программная платформа] // GitHub: [веб-сервис]. – GitHub, Inc., 2026. – URL: https://github.com/ollama/ollama.
7. Martin R. C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. – Boston: Pearson, 2017. – 432 p.
8. Berendeeva E. The Future of Space // The JetBrains Blog [блог-платформа]. –s.r.o., 2000. – URL: https://blog.jetbrains.com/space/2024/05/27/the-future-of-space/ (обновлено 27.11.2024).
9. Berendeeva E. SpaceCode On-Premises after May 31, 2025? // Space Support | JetBrains. – The JetBrains Blog [блог-платформа]. – JetBrains s.r.o., 2000-[2026]. – URL: https://space-support.jetbrains.com/hc/en-us/articles/19158813299218-Can-I-continue-using-Space-SpaceCode-On-Premises-after-May-31-2025 (обновлено 26.11.2024).
10. Bass, L. DevOps: A Software Architect's Perspective / L. Bass, I. Weber, L. Zhu. – Boston: Addison-Wesley, 2015. – 352 p.

11. Humble, J. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation / J. Humble, D. Farley. – Boston: Addison-Wesley, 2010. – 512 p.

12. Forsgren, N. Accelerate: The Science of Lean Software and DevOps / N. Forsgren, J. Humble, G. Kim. – Portland: IT Revolution Press, 2018. – 288 p.

13. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks / P. Lewis, E. Perez, A. Piktus [et al.] // Advances in Neural Information Processing Systems. – 2020. – Vol. 33. – P. 9459–9474.