ChatGPT

QIKI OS: Архитектура виртуальной операционной системы

Цель системы QIKI OS (виртуального бота)

QIKI OS – программно-симулированная бортовая система автономного космического робота (бота) в рамках хардкорного игрового симулятора. Проект QIKI_DTMP задуман как реалистичная песочница космических операций: игрок выступает **Оператором**, а **QIKI** – автономным космическим ботом, который **реально летает, добывает ресурсы, участвует в боях и разведке** [1]. Оператор взаимодействует с ботом удалённо через терминал, посылая задания (“курс на астероид X”, “разведка”, “добыча”, “бой”), а QIKI выполняет команды, обладая собственной логикой и даже способностью предупреждать об опасных действиях [2]. Такая “цифровая модель” (digital twin) позволяет **симулировать, тестировать и мониторить физические объекты** (робота и его окружение) без риска для реального прототипа [3]. Система стремится к максимальному реализму поведения – например, планируется задействовать реальные физические движки (Box2D, Bullet) для расчёта динамики мира [4], – **однако** акцент сделан на функциональную достоверность вместо низкоуровневой эмуляции электроники. Иными словами, QIKI OS моделирует **результаты работы устройств** (сенсоры, приводы, связь) и высокоуровневую логику бота, не пытаясь побитно эмулировать каждое железо. Такой подход обеспечивает реалистичность симуляции, сохраняя при этом разумную сложность проекта.

Эволюция архитектуры: от “микроядра” к микросервисам

Изначальная концепция (микроядро): ранние разработки рассматривали QIKI OS как виртуальную ОС бота, работающую по принципам микроядерной архитектуры. В этой модели внутри одного процесса (или тесно связанного набора потоков) существует **минимальное ядро** с базовыми сервисами, а прочие функции вынесены в модули: драйверы устройств, менеджеры датчиков/приводов, FSM поведения и т.д. Такой монолитно-модульный подход позволял обращаться со всеми компонентами как с частью единой ОС бота, приближая симуляцию к реальной прошивке. **Плюсы:** целостность системы, минимальные накладные расходы на межмодульное взаимодействие, более “реальное” устройство (всё как бы работает на одном борту). **Минусы:** сложность отладки и риска гонок в одном процессе (проблемы, с которыми столкнулись в предыдущих прототипах `qiki_bot` и др.), смешение разных уровней логики, недостаток модульной изоляции. Например, в старом проекте все данные сваливались вместе – *“телеметрия, состояние (FSM) и статическая конфигурация часто смешивались в одних файлах”*, что признано серьезным анти-паттерном [5]. Ошибка одного компонента могла обрушить всё приложение, а расширяемость и тестирование были затруднены.

Новая версия (микросервисная платформа): в ходе архитектурного анализа было решено разделить функциональность QIKI OS на отдельные сервисы с чёткими интерфейсами. **QIKI Digital Twin Microservices Platform** – это модульная платформа для цифрового двойника, основанная на микросервисах [6]. Выделено несколько независимых сервисов, каждый со своей

зоной ответственности, взаимодействующих через легковесные протоколы и шину событий. В Phase 1 были реализованы ключевые компоненты в виде контейнеров Docker [7]:

- **Q-Sim Service** – симулятор физического мира и сенсоров (генератор данных окружения и датчиков) [8].
- **Q-Core Agent** – основной "мозг" бота: включает конечный автомат поведения (FSM), логику принятия решений (правила, нейросеть) и управляет исполнительными механизмами [8].
- **Q-Operator Console** – консоль оператора (CLI/TUI-интерфейс), через которую пользователь наблюдает и командует системой [9].
- Дополнительно: **NATS JetStream** брокер сообщений (событийная шина для телеметрии и команд), сервисы поддержки вроде **FastStream Bridge** (обработка сырых данных радара) и **Registrar** (аудит и логирование событий) [10].

Разделение на микросервисы даёт существенные преимущества: **независимое развертывание** и обновление частей системы, **масштабируемость** (например, можно запустить несколько экземпляров симулятора или мозга при необходимости), **изоляция отказов** (падение одного сервиса не обрушит всю систему) [11]. В проекте отмечено, что архитектура микросервисов с чётким разделением ответственности и event-driven взаимодействием стала одной из ключевых сильных сторон системы [12]. Например, BIOS (см. далее) вынесен в отдельный микросервис `q-bios-service`, что позволило изолированно тестировать и развивать загрузочные процедуры [13].

Однако микросервисный подход усложняет интеграцию: появляется распределённость, необходимость оркестрации сервисов и обмена сообщениями. Как прямо указано в документации, такой подход *"повышает гибкость системы, но усложняет управление распределёнными компонентами"* [3]. Нужно обеспечивать согласованность версий API, запуск нескольких процессов (например, через `docker-compose`), наблюдаемость сети сообщений и т.д. Эти сложности компенсируются строгим **контрактно-ориентированным подходом** (versioned API, Proto/JSON схемы) и мощной автоматизацией (тесты, мониторинг), заложенными в проекте [14] [15].

Итог эволюции: команда пришла к архитектуре, сочетающей лучшие идеи обеих парадигм. Логика бортовой ОС всё так же структурирована по слоям (boot, kernel, drivers, модули), **но** каждый слой при необходимости выносится в отдельный сервис или модуль, если это повышает надёжность и удобство разработки. В результате финальный дизайн представляет собой виртуальную ОС, распределённую на несколько взаимодействующих компонентов.

Финальная архитектура: компоненты и принципы дизайна

Boot (загрузка системы и BIOS)

Этап загрузки реализован отдельным сервисом **Q-BIOS** – аналогом BIOS/POST для виртуального бота. При старте симуляции BIOS получает управление первым и выполняет функции, схожие с обычным BIOS ПК [16]. Он **инициализирует и тестирует "оборудование"** бота (виртуальные устройства), после чего передаёт управление основной программе (ядру Q-Core) [16]. В проекте QIKI OS BIOS оперирует декларативным описанием конфигурации железа: при запуске *чтение спецификаций* из файла `bot_physical_specs.json` даёт список всех портов/устройств на борту [17]. Затем выполняется **Power-On Self-Test (POST)**: BIOS по очереди опрашивает каждый указанный компонент, выполняя базовые проверки связи и статуса [18]. Например, на этапе *[0.02s - 0.20s]* происходит проход по всем портам с запросом ответа, состояния и пр. [18]. Итогом является формирование отчёта `bios_status` со списком устройств и результатов тестов (OK/

ошибка) [19]. Если всё прошло успешно (`status: "ready"`), BIOS публикует событие `hardware_ready` и переходит в режим ожидания запросов диагностики; если выявлены неполадки (`status: "error"`), выдаётся событие `hardware_error` с кодами ошибок и BIOS остаётся активным для отладочных команд [19].

Отдельного внимания заслуживает **plug-and-play механизм** системы: состав оборудования полностью определяется конфигурацией. Добавить новый сенсор или мотор можно, просто дописав его параметры в `bot_physical_specs.json` – BIOS при следующем запуске сам его обнаружит и протестирует [17] [20]. Таким образом достигается гибкость "горячего" подключения модулей без изменения кода. Для каждого порта указывается тип устройства и протокол (например, `lidar` по виртуальной шине `sim_bus` или реальный I^2C) [21]. Это закладывает основу для будущей интеграции с реальным железом: документ открытых вопросов BIOS рассматривает возможность реализации `hardware_adapter` для подключения физической периферии вместо симулятора [22]. Но в текущей версии всё оборудование эмулируется, а BIOS служит единым интерфейсом доступа. Архитектурно Q-BIOS сделан **максимально простым** и изолированным: он не содержит бизнес-логики поведения, не знает про FSM или задачи бота – только низкоуровневая инициализация и предоставление API статусов [23]. Высокоуровневый агент (ядро) никогда не лезет к "железу" в обход BIOS – это запрещено архитектурой (анти-паттерн "обход BIOS") [23]. Такой дизайн повторяет принцип реальных ОС: сначала выполняется BIOS/bootloader, который готовит оборудование и затем передает контроль ядру.

Kernel (ядро Q-Core Agent)

Q-Core Agent – центральный сервис, играющий роль *ядра виртуальной ОС* и "мозга" бота. После успешного POST от BIOS, ядро берет на себя управление. Логически Q-Core разбит на несколько основных частей: - **Bot Core** – низкоуровневый модуль ядра, отвечающий за идентификацию бота, загрузку его конфигурации и абстракцию доступа к сенсорам/актуаторам [24] [25]. Он хранит статические свойства (например, геометрию, лимиты скорости, ёмкость батареи) и уникальный идентификатор робота [26], и предоставляет унифицированные точки ввода-вывода для необработанных данных (*SensorRawIn*, *ActuatorRawOut*). *Bot Core* отделён от поведенческой логики (FSM) – это именно "зерно" системы, инкапсулирующее аппаратный профиль, чтобы выше по стеку оперировать абстракциями. - **FSM Handler** – подсистема конечного автомата состояний, управляющего поведением бота. FSM определяет основные режимы работы (BOOT, IDLE, ACTIVE, SAFE_MODE и пр.) и переходы между ними. При старте ядра FSM находится в состоянии BOOT и ожидает сигналов от BIOS. После получения `hardware_ready` FSM переключается в **IDLE (готовность к работе)** [27]. Далее FSM реагирует на события мира и команды оператора, включая переходы в другие состояния (например, активное выполнение миссии, аварийный безопасный режим при сбоях и т.д.). - **Rule Engine и Neural Engine** – движки принятия решений. Rule Engine содержит жёстко запрограммированные правила и рефлексы (например, если BIOS сообщил об ошибке – перейти в SAFE_MODE) [28]. Neural Engine – задел под интеллектуальные алгоритмы (машинное обучение), пока что в виде заглушки, генерирующей тестовые "предложения" действий [29]. Результаты обоих оцениваются модулем **Proposal Evaluator/Arbiter**, который выбирает финальное действие бота на каждом цикле тикрования. - **Tick Orchestrator** – координатор цикла обновления. Q-Core работает дискретными тактами (например, каждые 5 секунд) [28]. В каждом цикле собираются новые данные от сенсоров, FSM обновляет состояние, Rule/Neural Engine генерируют кандидаты действий, Arbiter выбирает действие, и итоговые команды отправляются актуаторам. Затем цикл повторяется. Такая *step-based архитектура с конфигурируемым тиком* зафиксирована как правильное решение проекта [30], обеспечивая детерминизм и воспроизводимость симуляции.

Ядро Q-Core тесно взаимодействует с симулятором мира (Q-Sim) через четко определенные интерфейсы. *Bot Core* как раз и выступает прослойкой: он получает “сырые” данные от сенсоров из Q-Sim и передает “сырые” команды на актуаторы обратно в симуляцию [31]. На диаграмме дизайна это показано так: SensorSim (в Q-Sim) посылает пакет SensorRawIn в Bot Core, а Bot Core отправляет ActuatorRawOut в ActuatorSim (модель двигателей) [31]. Таким образом, ядро вообще не знает, реальное ли оборудование или симуляция – оно работает с абстракциями сенсоров и приводов. Сами же вызовы осуществляются либо по gRPC (для запросов данных, команд) либо по шине событий (асинхронные уведомления). Например, в Phase 1 *Q-Core Agent* запрашивает у *Q-Sim Service* данные сенсоров через метод `GetSensorData` (gRPC), а события от симуляции (например, радарные контакты) приходят через NATS JetStream [10].

Отдельно стоит упомянуть, что ядро загружает и **динамические модули “корабля”** – расширения, описывающие специфические подсистемы. В кодовой базе присутствуют файлы `ship_*.py` [32] и `ship_config.json` [33]: это задел на управление бортовыми системами вроде двигателей ориентации, стыковочных узлов, радиосвязи и т.п. (фаза **Step-A** развития). Эти модули подключаются к ядру при старте (загрузка конфигурации корабля, инициализация контекста AgentContext) [34]. Принцип **plug-and-play** здесь тоже реализован: благодаря декларативным конфигам BotSpec и ship_config, ядро может активировать или игнорировать те или иные подсистемы в зависимости от их наличия. Хеширование “профиля оборудования” контролирует версионность конфигурации [35] – если состав устройств изменился, система это распознает. Всё это делает архитектуру QIKI OS очень **модульной**: ядро выступает платформой, на которую можно навешивать новые функциональные блоки без ломки остального.

Драйверы устройств и взаимодействие с симуляцией

В силу распределённости архитектуры, понятие **“драйверы”** принимает особую форму. Нет традиционных драйверов в ядре, работающих с железом через регистры – вместо этого функцию драйверов выполняют **адаптеры протоколов и API**, связывающие Q-Core и Q-Sim. Каждое виртуальное устройство (порт) имеет указанный протокол взаимодействия (например, `sim_bus` для симулируемого лидара) [36]. BIOS и Bot Core вместе реализуют своеобразный HAL (Hardware Abstraction Layer): - BIOS на этапе инициализации выясняет, какие устройства присутствуют и работают, - Bot Core хранит информацию о них и предоставляет методы для чтения/записи.

После завершения загрузки **ядро Q-Core обращается к симулятору через стандартизированные вызовы**. Например, запрос данных лидара может идти через gRPC метод `GetSensorData(LIDAR)` у Q-Sim, который вернёт текущее значение дальности [37]. Аналогично, команда двигателю – вызов `SetVelocity` с параметром, который Q-Sim применит к своей физической модели [38]. В архитектуре предусмотрено, что все обмены стандартизованы Protobuf контрактами: определены универсальные типы данных сенсоров и команд (со временем, единицами измерения и т.д.), так что “драйвер” в Q-Core просто формирует/парсит эти сообщения [39] [40].

Для асинхронных данных используется **событийная шина (event bus)**. Здесь ключевую роль играет **NATS JetStream** – центральный брокер, через который летят телеметрия, события и команды. Такая шина выполняет ту же задачу, что шина ввода-вывода в ОС: доставляет сообщения от одних компонентов другим, только по сети. Выделены стандартные темы (subjects) NATS: например, все **телеметрические сообщения** публикуются в канал `qiki.telemetry`, все **события** – в `qiki.events.v1.*` [41]. Благодаря этому любой сервис, подписавшийся на эти темы (будь то операторская консоль или отладочный логгер), получает поток событий в реальном времени. Доставка настроена по принципу *at-least-once* с хранилищем JetStream, чтобы не терять данные [42]. Кроме того, вводится единый формат событий (CloudEvents) – каждый JSON/

Protobuf снабжён заголовками типа `ce_type`, `ce_time` и пр., что стандартизирует обмен [43] [44]. В целом, **событийная архитектура** позволяет слабо связать компоненты: Q-Core публикует, к примеру, событие о смене состояния FSM, а операторский интерфейс просто отображает его, не вызывая напрямую методов ядра.

Резюмируя, финальная архитектура действует подобно реальной ОС, но распределённой: **Boot/BIOS** готовит "железо" и проверяет его, **Kernel/Q-Core** управляет состояниями и решает, что делать, **Drivers/HAL** (BIOS+Core) обеспечивают обмен данными с уровнем мира, **Modules** расширяют функциональность особыми подсистемами, а **Event Bus** связывает всё воедино, гарантируя событийное, plug-and-play взаимодействие компонентов без жёсткой привязки друг к другу.

Операторская консоль ORION (интерфейс пользователя)

ORION – это высокоуровневый компонент, представляющий "лицо" системы для пользователя. Архитектурно ORION – отдельный сервис (Q-Operator Console), подключающийся к QIKI OS через описанные API и шину событий. Он реализован как текстовый UI (*TUI*) на базе библиотеки Textual, работая в терминале для атмосферы ретро-командного центра [45]. ORION выступает своего рода **"оболочкой OS"** для оператора, имитируя консоль космического аппарата. Интерфейс разделён на несколько экранов (системы, радар, события, консоль команд, сводка, питание, диагностика, управление миссией и т.д.) – все они интегрированы в единое приложение с общей хром-рамкой (шапка, боковая навигация, строка ввода команд) [45] [46]. В любой момент доступна глобальная шина команд, куда оператор может вводить инструкции для бота [47].

Связь ORION с остальными компонентами полностью построена на внешних интерфейсах – никаких скрытых вызовов внутрь Q-Core. Консоль подключается к **NATS**: она подписывается на потоки телеметрии и событий (`qiki.telemetry`, `qiki.events.v1.>` и др.) и тем самым **получает всю информацию о состоянии бота и мира в режиме реального времени** [41]. Например, приходящие телеметрические сообщения (состояние систем, координаты, скорость, напряжение и пр.) сохраняются во **внутренний снэпшот хранилища ORION** – *SnapshotStore*, который отслеживает "последнее известное состояние" по каждому типу данных [48]. Это позволяет UI показывать актуальные значения на экранах "Сводка", "Система питания", "Диагностика" и помечать их как *СВЕЖО/УСТАРЕЛО/НЕТ* по встроенным порогам времени давности [49]. События (частые потоки, близкие к телеметрии) отображаются на экране "События" с возможностью фильтрации по типу, источнику, уровню важности [50]. Любой выполненный оператором **командный ввод** отправляется в виде сообщения: ORION парсит строку (поддерживается билингвальность команд на `EN/RU` и алиасы) и публикует соответствующий сигнал, например, команда `simulation.pause` будет опубликована на control-шину `qiki.commands.control` [51]. Далее по этому каналу Q-Core Agent или Q-Sim получат команду и выполнят действие (например, приостановят симуляцию). Обратная связь (результат команды, отчёт) приходит как ответное сообщение и отображается в панели "Консоль" [52].

Таким образом, ORION соблюдает принцип разделения: **никаких прямых вызовов**, только обмен сообщениями. Это делает интерфейс нечувствительным к внутреннему устройству – он не "знает", микросервисы там или монолит, для него QIKI OS – это набор стандартных каналов данных. Данный дизайн обеспечивает гибкость (можно хоть вовсе отключить UI – на работу симуляции это не повлияет) и соответствует философии *"no-mocks"*: интерфейс не показывает ничего, чего реально нет в данных [53]. Если какой-то датчик отсутствует, на экране будет `N/A/НД`, никаких вымышленных значений. ORION, по сути, играет роль командно-контрольной подсистемы поверх виртуальной ОС, завершая общую архитектуру "бот + оператор".

Жизненный цикл системы (от запуска до взаимодействия с оператором)

Полный цикл работы QIKI OS можно представить как последовательность фаз, каждая из которых отвечает определённому компоненту архитектуры:

1. **Инициализация и загрузка BIOS:** при старте симуляции (например, запуском `docker-compose` Phase 1) первым поднимается сервис `q-bios-service`. Это эквивалент подачи питания на бот – BIOS начинает POST. Он быстро считывает профиль оборудования, опрашивает все указанные устройства и формирует отчёт о готовности [17] [19]. Лог-файл BIOS фиксирует этапы: запуск BIOS, чтение профиля (с хешем для контроля версии), тест каждого порта с отметкой OK или кодом ошибки, завершение POST со статусом [54]. Например, может быть зафиксировано: *"[0.05s] POST: Probing port motor_left... OK"*, *"[0.07s] ... imu_main... ERROR (Code: 0x02)"* и т.д. [54]. В конце, допустим, BIOS видит, что один из датчиков не в норме – тогда статус = ERROR, код 0x02 (нестабильные показания) [55] [56]. BIOS публикует событие `hardware_error` с деталями и переходит в режим диагностики, ожидая команд оператора для перезагрузки или попытки исправить ситуацию [19]. Если же всё "зелёное", BIOS сообщает `hardware_ready` и становится пассивным.

2. **Запуск ядра Q-Core и переход в рабочее состояние:** параллельно с BIOS, запускается основной агент **Q-Core**. При инициализации он выполняет несколько шагов: **(1)** загружает конфигурацию бота (`bot_config.json` и при наличии `ship_config.json`) – в них описаны параметры модели, режим (например, `mode: "full"` или минимальный) и т.д.; **(2)** генерирует или считывает сохранённый уникальный ID бота (например, `QIKI-20250721-a1b2c3d4`) [26]; **(3)** создаёт **контекст AgentContext** – централизованную структуру, хранящую текущее состояние всех подсистем (сюда будут стекаться данные BIOS, FSM, и пр.); **(4)** готовится к цикличной работе, инициализируя таймеры, TickOrchestrator и пр. [34]. Пока Q-Core находится в состоянии **BOOT** и ждет сигнала от BIOS. Как только BIOS публикует `hardware_ready`, агент получает либо прямое уведомление (через API вызов `get_bios_status()`), либо сообщение по шине. Убедившись, что все системы в порядке, Q-Core выполняет *первый переход FSM*: **BOOT → IDLE** [27]. Теперь бот официально "просыпается" и готов к выполнению своих задач. Ядро отправляет подтверждение (например, логирует в `action.log` событие о завершении загрузки) [57]. Если вместо готовности пришёл сигнал об ошибке, FSM перейдёт в **SAFE_MODE** – специальный режим, при котором бот не начинает активных действий, а ожидает решения проблемы [58]. Этот сценарий тоже предусмотрен: Q-Core может зафиксировать *"Emergency protocols activated"* и остаться в безопасном состоянии, пока оператор не вмешается [58].

3. **Основной цикл симуляции (такты выполнения):** перейдя в IDLE, Q-Core начинает бесконечный цикл тиków. Каждые N секунд происходит обновление:

4. **Запрос данных от мира:** агент получает свежие показания сенсоров. Например, дергается gRPC метод `GetSensorData` симулятора, возвращающий структуру SensorRawIn для каждого датчика (лидар, IMU и т.п.) [38]. Если подключён радар, агент может подписаться на поток сообщений от FastStream (через NATS JetStream) и получать пакеты с метками целей [59]. Все эти сырые данные записываются в **Runtime Buffer** Bot Core [60] [61] – т.е. локально сохраняются последние значения. Одновременно они транслируются выше: FSM и другие компоненты могут их использовать для принятия решений.

5. **Обновление состояний и принятие решений:** на основании обновлённых данных FSM проверяет, не пора ли сменить состояние (например, если обнаружен враг – переход в боевой режим). Rule Engine генерирует реакции: например, правило “батарея разряжена” может выставить предложение вернуться на базу. Neural Engine (если активен) выдает предложения действий, основанные на обученных моделях (пока заглушки). Все предложения оцениваются, и **Arbiter/Proposal Evaluator** выбирает финальное действие этого цикла – например, *“включить двигатели на курс к точке X”*.

6. **Выполнение действий (выход команды):** выбранная команда формируется в структуру ActuatorRawOut (например, `{ actuator_id: "motor_left", command: "set_velocity_percent", value: 50 }` для 50% тяги на левый мотор) [62] [63]. Через Bot Core эта команда отправляется в симуляцию: либо вызывается соответствующий RPC метод Q-Sim (например, `ApplyCommand(motor_left, 50%)`), либо публикуется сообщение в шину команд `qiki.commands.control` [64], на которое подписан симулятор. Q-Sim применяет команду к модели – обновляет скорость объекта. Если действие подразумевает длительный эффект (например, поворот на 90°), FSM может перейти в промежуточное состояние (EXECUTING) и дождаться события завершения (которое сгенерирует симуляция, например, событие *“rotation_done”* в `qiki.events.v1.mission`) прежде чем перейти дальше.

7. **Логирование и метрическая телеметрия:** каждое критичное действие и событие фиксируется. Сервис Registrar записывает структурированные логи в `action.log` с указанием времени, кода события, типа действия и пр. [65]. Параллельно обновляются метрики (Prometheus): например, число активных целей радара, уровень предупреждений Guard-системы, лаги очередей JetStream и пр. (по состоянию на конец Stage 0 были реализованы метрики `qiki_agent_radar_active_tracks`, `qiki_agent_guard_critical_active` и др. [66]). Это позволяет в реальном времени мониторить здоровье системы.

По окончании цикла ядро ждёт следующего тик-интервала и повторяет процесс. В течение всего цикла Q-Core также может реагировать на **внешние команды**: если оператор ввёл что-то через консоль (например, *“screen radar”* или команду изменения режима), ORION опубликует соответствующее сообщение, которое поступит агенту. Q-Core обработает команду (возможно, через отдельный ControlHandler) – скажем, команда *“sim.pause”* заставит симулятор приостановить обновление мира, о чём агент узнает через ответ или событие.

1. **Работа симулятора мира (Q-Sim) параллельно ядру:** сервис **Q-Sim Service** в это время выполняет свой цикл – расчёт физической модели. Он хранит в **WorldModel** состояние окружения: координаты бота, его скорость, уровень заряда, состояние датчиков и т.д. [38]. При получении команд от Q-Core (движение, поворот, активация инструмента) симулятор обновляет параметры модели. Каждые заданные интервал времени он генерирует новые значения сенсоров. Например, для простоты Phase 1 *“это лишь скалярное значение x-координаты”* [67] – т.е. симулятор отдает агенту текущее X-координату как датчик. В более продвинутой версии: вычисляет дистанции до препятствий (LIDAR), отслеживает параметры двигателя, может симулировать сбоевое поведение. Также Q-Sim отвечает за генерацию **радара** – отдельный компонент `q-sim-radar` публикует кадры обзора в несколько NATS-топиков (LR и SR диапазоны) [68], которые потом преобразуются FastStream Bridge в списки целей (tracks) и тоже поступают по шине. В результате, Q-Core получает “картинку” мира через события, а не прямым вызовом тяжелых вычислений.

2. **Подключение и работа операторской консоли:** в любой момент после запуска Phase 1 стека разработчик/пользователь может запустить **ORION** (например, командой Docker Compose для `qiki-operator-console`) [69]. Консоль соединяется с брокером NATS (URL берётся из переменных окружения, например, `NATS_URL=nats://qiki-nats-phase1:4222` в Docker) [70] и начинает получать потоки данных. ORION подписывается как минимум на:

3. **Telemetry** – канал `qiki.telemetry` для общей телеметрии систем [71]. Сюда Q-Core (или Registrar) отправляет регулярные статусы: например, проценты энергии, координаты, температуру систем и пр.
4. **Events** – шаблон `qiki.events.v1.>` (все события v1) [72]. Тут идут пакеты вроде `mission.task.started`, `guard.alert.raised`, `bios.hardware_error` и т.д. Операторский интерфейс фильтрует и группирует их по типам, позволяя просмотреть хронологию событий.
5. **Radar** – топики радара (если включены): `qiki.radar.v1.tracks`, `qiki.radar.v1.frames` и пр., хотя в документации оговорено, что радар UI не в приоритете и пока стабилизируется без активной функциональности [73].
6. **Control (Commands/Responses)** – каналы `qiki.commands.control` и `qiki.responses.control` для отправки команд и получения ответа на них [74]. ORION при вводе команды публикует сообщение в `commands.control`, а слушает `responses.control` чтобы отобразить результат (подтверждение или ошибку).

Получая данные, ORION обновляет свой интерфейс: значение телеметрии попадают в поля на соответствующих экранах (с отметкой свежести, которая меняется на *УСТАРЕЛО*, если давно не было новых сообщений) [49]. События текут в лог на экране *Events*, помеченные цветом по уровню и фильтруемые. В верхней строчке интерфейса отображается индикатор ONLINE (связь с ботом) и ключевые показатели здоровья [46] – они вычисляются на основе последних телеметрических сообщений (например, связь считается потерянной, если ничего не было X секунд). Оператор может переключаться между экранами (F1–F8 привязаны к System, Radar, Events, Console, Summary, Power, Diagnostics, Mission соответственно) [75], *но это не влияет на работу самого бота*, так как UI лишь подписчик.

Когда оператор вводит команду (скажем, *“двигатель_левый 50”* – установить 50% тяги левого двигателя), ORION распознает ее и транслирует как унифицированное сообщение. Q-Core, будучи подписанным (напрямую или через Registrar) на `qiki.commands.control`, получает команду, преобразует в вызов к симулятору (SetVelocity) и затем пишет результат в `qiki.responses.control`. ORION поймает ответ и выведет его на экран *Console* вместе с отметкой времени [52]. Таким образом, интерактивный цикл *“Оператор → ORION → QIKI OS → ORION → Оператор”* замыкается.

1. **Завершение работы и перезапуск:** по завершении миссии или по команде оператора система может быть остановлена. Обычно это делается командами Docker (`docker compose down`), что корректно тушит все сервисы. В рамках самой симуляции предусмотрены мягкие сценарии: Q-Core может перейти в состояние SHUTDOWN, оповестив оператора и остановив отправку команд симуляции, затем вызвать API BIOS `soft_reboot()` или инициировать сброс. BIOS, получив команду перезагрузки, может выполнить цикл POST заново (возможно, уже с новым конфигом – например, если “реплицируется” бот после гибели, меняются некоторые параметры) [76]. В лоре игры “смерти нет” – бот возрождается на станции с базовым профилем, а потерянные данные считаются ценой ошибки [76]. QIKI OS поддерживает такую механику: можно перезапустить BIOS/ядро без перезапуска всего контейнерного окружения (к примеру,

отправив `hot_reload_config()` BIOS-у для перечитывания спека или полностью перезапустив сервисы). Благодаря логированию и хранению ID бота, после рестарта система распознает “тот же” это бот или новый экземпляр (если новая сессия – сгенерируется новый ID и начнётся как будто с чистого листа) [26] . При последовательном подходе к обновлению (в проекте описаны *RESTART_CHECKLIST* и протоколы перезапуска) можно безопасно перенести контекст, если это задумано.

Таким образом, жизненный цикл QIKI OS повторяет цикл реального робота: запуск → самотестирование → дежурство → активность/миссия (с регулярным обновлением по тактам) → приостанока или сбой → потенциальный перезапуск. Все этапы документированы и сопровождаются телеметрией, что позволяет разработчикам и операторам отслеживать работу системы на каждом шаге. Финальная архитектура и дизайн QIKI OS сбалансированы между технической точностью и модульностью: каждый компонент имеет чёткую ответственность, общается через задокументированные интерфейсы и события, а вместе они образуют цельную виртуальную операционную систему для хардкорного симулятора космического бота.

Источники (из репозитория QIKI_DTMP): архитектурные документы и отчёты разработки, включая стратегический план [77] [8] , актуальное описание Phase 1 архитектуры [78] , дизайн BIOS [17] [19] , ядра бота [24] [31] , а также рабочие заметки и лор проекта [1] [79] , послужили основой для данного свода. Каждый фрагмент, цитируемый в тексте, напрямую взят из этих файлов, отражая реальные решения (и изменения) архитектуры QIKI OS на пути от концепции к реализованной системе.

---

[1] [2] [76] [79] qiki_operator_lore_notes.md
https://github.com/sonra44/QIKI_DTMP/blob/7637b6824263dd9bed7eda2e8454121f848064b3/docs/design/game/qiki_operator_lore_notes.md

[3] [67] FILEDEVELOPER.md
https://github.com/sonra44/DTMP/blob/f0843b3dca263f6a2f8fd040b2f8395b3f02275a/QIKI_DTMP/FILEDEVELOPER.md

[4] [8] [9] [77] NEW_QIKI_PLATFORM_DESIGN.md
https://github.com/sonra44/QIKI_DTMP/blob/7637b6824263dd9bed7eda2e8454121f848064b3/docs/NEW_QIKI_PLATFORM_DESIGN.md

[5] [24] [25] [26] [31] [60] [61] [62] [63] bot_core_design.ru.md
https://github.com/sonra44/QIKI_DTMP/blob/7637b6824263dd9bed7eda2e8454121f848064b3/docs/design/q-core-agent/bot_core_design.ru.md

[6] [11] [12] [14] [27] [30] [32] [33] [34] [35] [39] [40] [57] [58] [65] CLAUDE_PROJECT_ANALYSIS.md
https://github.com/sonra44/QIKI_DTMP/blob/7637b6824263dd9bed7eda2e8454121f848064b3/CLAUDE_PROJECT_ANALYSIS.md

[7] [10] [68] [78] README_OLD.md
https://github.com/sonra44/QIKI_DTMP/blob/7637b6824263dd9bed7eda2e8454121f848064b3/README_OLD.md

[13] [16] [17] [18] [19] [20] [21] [22] [23] [36] [54] [55] [56] bios_design.md
https://github.com/sonra44/QIKI_DTMP/blob/7637b6824263dd9bed7eda2e8454121f848064b3/docs/design/q-core-agent/bios_design.md

[15] [42] QIKI_DTMP_METHODOLOGICAL_PRINCIPLES_V1.md
https://github.com/sonra44/QIKI_DTMP/blob/7637b6824263dd9bed7eda2e8454121f848064b3/docs/QIKI_DTMP_METHODOLOGICAL_PRINCIPLES_V1.md

28 29 38 TASK_20250805_FULL_PROJECT_ANALYSIS.md
https://github.com/sonra44/QIKI_DTMP/blob/7637b6824263dd9bed7eda2e8454121f848064b3/TASKS/TASK_20250805_FULL_PROJECT_ANALYSIS.md

37 43 44 59 66 CURRENT_STATE.md
https://github.com/sonra44/QIKI_DTMP/blob/7637b6824263dd9bed7eda2e8454121f848064b3/CONTEXT/CURRENT_STATE.md

41 45 46 47 48 49 50 51 52 53 64 69 70 71 72 73 74 75 ORION_OS_SYSTEM.md
https://github.com/sonra44/QIKI_DTMP/blob/7637b6824263dd9bed7eda2e8454121f848064b3/docs/design/operator_console/ORION_OS_SYSTEM.md