ChatGPT

Финальный технический отчёт: QIKI OS и ORION

Введение

Проект **QIKI Digital Twin Microservices Platform (QIKI DTMP)** направлен на создание виртуального двойника автономного космического бота (QIKI). Он сочетает **игровой симулятор** и **реалистичную модель бортовой операционной системы**, позволяя игроку работать с роботом как с реальным космическим кораблём. В процессе анализа репозитория `sonra44/QIKI_DTMP` были изучены ключевые проектные документы, описывающие архитектуру, жизненный цикл системы, интерфейс оператора и планы развития. Ниже представлена сводка, объединяющая найденную информацию.

Эволюция архитектуры

Проект прошёл путь от **микроядерной ОС в одном процессе** до **распределённой микросервисной платформы**. Первоначальная концепция предполагала, что QIKI OS будет работать как классическая ОС робота: минимальное ядро, набор драйверов и модулей, единая шина сообщений. Это обеспечивало целостность, но усложняло тестирование и расширение. В ходе разработки архитектура была пересмотрена — ответственность разделили между несколькими сервисами, взаимодействующими через NATS JetStream и gRPC.

В результате Phase 1 были выделены основные контейнеры: - **qiki-nats-phase1** (брокер сообщений), **q-sim-service** (симулятор физики и сенсоров), **q-sim-radar** (генерация радарных кадров), **faststream-bridge** (валидация и преобразование данных), **qiki-dev** (агент/ядро), и **nats-js-init** (инициализация потоков и durable-консьюмеров) [1].
- Порядок их запуска и необходимость проверки здоровья для каждого сервиса описаны в `ARCHITECTURE.md` [1].

Эта модульность позволила изолировать ошибки, масштабировать отдельные части и упростить подключение новых компонентов (например, модуль радара или другой симулятор). Недостатки старой схемы — смешение конфигурации и состояния, отсутствие чётких контрактов и невозможность «горячего» подключения устройств — были устранены через строгие API, схемы сообщений и разделение на сервисы.

Boot/BIOS: старт системы и проверка оборудования

Первыми в цепочке инициализации работают **BIOS и boot-layer**. Сервис `q-bios-service` выполняет функции POST: читает спецификацию бота (`bot_physical_specs.json`), последовательно тестирует каждый порт, формирует отчёт `bios_status.json` и публикует событие `hardware_ready` или `hardware_error` [2] [3].

BIOS предоставляет API: * `get_bios_status` – возвращает полный отчёт, * `get_component_status` – информацию о конкретном порте, * `soft_reboot` и `hot_reload_config` – перезагрузка и перечитывание конфигурации【48†L75-L82】.

Архитектура запрещает бизнес-логику в BIOS; весь контроль переходит ядру после успешного POST. Важный элемент — **plug-and-play**: новый сенсор или двигатель добавляется через описатель в `bot_physical_specs.json`, BIOS автоматически его обнаружит и протестирует【48†L43-L51】. Такие устройства будут «подключаться» как модуль, драйвер которого реализует стандартный контракт.

Ядро Q-Core Agent

Главный микросервис — **Q-Core Agent** — отвечает за «мозг» бота. Его архитектура состоит из нескольких модулей:

1. **Bot Core** — низкоуровневый компонент, хранящий статическую конфигурацию робота (уникальный ID, геометрию, лимиты тяги и т.п.) и предоставляющий унифицированные API для чтения/записи сенсоров и актуаторов【55†L12-L20】【55†L45-L53】. В нём находится **runtime buffer** с последними данными сенсоров/приводов【55†L25-L33】. Анти-паттерны включают смешивание состояния и конфигурации, отсутствие устойчивого ID, прямой доступ к оборудованию и монолитные getter'ы【55†L39-L44】.

2. **FSM Handler** — конечный автомат, задающий основные режимы работы: BOOT, IDLE, ACTIVE, SAFE_MODE и др. После сигнала `hardware_ready` FSM переходит из BOOT в IDLE и запускает обновление тактов【25†L83-L90】.

3. **Rule Engine и Neural Engine** — два источника «предложений» (proposals) по действиям. Rule Engine выдаёт приоритетные действия на основе жёстко заданных правил; Neural Engine (пока задел) — вероятностные предложения с коэффициентом уверенности [4]. Оба модуля могут перезагружаться на лету для обновления логики.

4. **Proposal Evaluator/Arbiter** — компонент, который получает предложения, фильтрует их, сортирует по приоритету и выбирает финальную команду (например, установить тягу двигателей) [4].

5. **Tick Orchestrator** — координирует цикл обновлений: собирает новые данные сенсоров, обновляет FSM, запускает Rule/Neural Engine, вызывает Arbiter и отправляет команды в симулятор. Этот цикл проходит с заданным интервалом (напр. 5 с) [1].

Взаимодействие ядра с симулятором идёт через gRPC: методы `GetSensorData`, `SendActuatorCommand`, `GetRadarFrame` и другие, определённые в `ARCHITECTURE.md` [1]. События и телеметрия публикуются в NATS, что реализует асинхронную шину данных.

Симулятор Q-Sim и радар

Q-Sim Service — независимый сервис, эмулирующий физику и сенсоры. Его функции: * хранить состояние мира (WorldModel), * рассчитывать движения и отдавать новые значения сенсоров (пока в Phase 1 это один скаляр X-координаты) [1], * принимать команды актуации (скорости двигателей и т.п.) и применять их к модели.

Параллельно работает модуль **Q-Sim Radar**, генерирующий дальномерные кадры и передающий их в поток `qiki.radar.v1.frames`. Эти данные проходят через **FastStream Bridge**, который валидирует, преобразует и публикует сообщения `qiki.radar.v1.tracks` [1]. Симулятор и мост работают в собственном тактовом цикле и не блокируют ядро.

Event Bus: обмен сообщениями и телеметрия

Центральной связующей тканью системы является **NATS JetStream**. Все сервисы (BIOS, Q-Core, Q-Sim, ORION) публикуют и подписываются на события по определённым темам (subjects).

- Телеметрия публикуется в `qiki.telemetry` — здесь идут параметры энергии, температуры, координаты и т.п.
- События (например, `mission.task.started`, `guard.alert.raised`, `bios.hardware_error`) — в шаблон `qiki.events.v1.>` 【10†L43-L52】.
- Команды управления поступают в `qiki.commands.control`, а ответы на них — в `qiki.responses.control` 【10†L61-L65】.
- Радарные кадры и треки — в `qiki.radar.v1.frames` и `qiki.radar.v1.tracks` [1].

Каждое сообщение следует стандарту CloudEvents: имеет метки времени (`ce_time`), типа, источника и полезную нагрузку, что упрощает маршаллинг и обработку на стороне различных сервисов【12†L19-L24】.

Plug-and-Play модули и расширение

Для поддержки расширяемости используются **стандартизованные интерфейсы модулей**. Разработчик может создать новый прибор (например, плазменный ускоритель) как отдельный сервис или библиотеку, реализующую контракт: методы `initialize()`, `update(delta_time)`, `shutdown()`, описание входных/выходных параметров и структуру событий. При подключении ядро сканирует папку модулей, считывает метаданные и регистрирует новый модуль в системе — принцип plug-and-play, упрощающий добавление устройств. В дизайне подчёркивается, что модули моделируют **реалистичное поведение** без полной эмуляции электроники: они получают входные воздействия (питание, команды), вычисляют результаты (тяга, расход энергии) и публикуют данные на шину.

Пользовательский интерфейс ORION

Верхним уровнем системы выступает **ORION Shell OS** — текстовый интерфейс оператора, реализованный на библиотеке Textual. Он подключается к NATS и отображает данные в реальном времени. В документе `ORION_OS_SYSTEM.md` описаны:

- **Структура интерфейса**: шапка (header) с ключевыми показателями (заряд, состояние корпуса, температура), боковая навигация (Sidebar) со списком экранов, область инспектора (Inspector) для выбора и просмотра деталей объекта, и строка ввода команд 【10†L8-L19】【10†L103-L111】.

- **Экранные режимы**: System (сводные системы), Radar (таблица треков), Events (журнал событий с фильтром), Console (вывод команд и ответов), Summary (краткое состояние), Power (энергетика), Diagnostics (диагностика), Mission (цели)【10†L8-L19】.

- **Командная строка**: позволяет вводить команды (например, `screen radar`, `filter power`), которые публикаются в `qiki.commands.control` 【10†L133-L141】.

- **Правила UX**: интерфейс должен быть билингвальным (англ/рус), без аббревиатур, с устойчивым расположением элементов (stable chrome), а все данные должны быть реальными — никаких заглушек (политика «No Mocks») 【10†L16-L19】.

- **Панель инспектора**: отображает подробности выбранного объекта (тип, источник, ключ, возраст, preview JSON), что помогает оператору быстро ориентироваться 【10†L84-L92】.

В рабочем цикле ORION подписывается на телеметрию, события и данные радара, отображает их в таблицах, реагирует на выбор строки (Highlight → Inspector) и отправляет команды. Оператор видит, как QIKI исполняет миссию, без непосредственного взаимодействия с реализацией агентов.

Жизненный цикл системы

1. **Запуск и POST**: при запуске контейнеров `nats-js-init`, `q-sim-service`, `faststream-bridge`, `qiki-dev` и `q-sim-radar` поднимают инфраструктуру; BIOS запускается, читает спецификацию оборудования, проверяет порты и отправляет сигнал `hardware_ready` или `hardware_error` [1] [2].

2. **Инициализация ядра и FSM**: Q-Core Agent получает сигнал готовности, считывает конфигурацию и переходит из BOOT в IDLE. Он затем инициализирует контекст AgentContext, запускает таймеры и в цикле собирает данные сенсоров, генерирует предложения действий, выбирает команду и отсылает в симулятор. Если BIOS сообщает об ошибке — FSM переключается в SAFE_MODE и ждёт исправления 【25†L79-L87】.

3. **Симуляция мира и радар**: Q-Sim обновляет физику, публикует новые значения сенсоров; радарный сервис генерирует кадры, которые через FastStream Bridge трансформируются в треки и публикуются в NATS [1].

4. **Реакция на события и команды**: вся телеметрия и инциденты пересылаются через шину. Q-Core, увидев событие от мира (например, контакт радара), может сменить режим или сгенерировать новую команду; оператор через ORION вводит команды, которые также проходят через шину и обрабатываются соответствующим сервисом 【10†L61-L65】.

5. **Наблюдение оператором**: ORION отображает состояние систем, миссий, энергии и событий; оператор может фильтровать журнал, просматривать подробности в инспекторе, запускать/останавливать симуляцию, изменять сценарии. Все взаимодействия реализованы через события и команды, интерфейс не зависит от конкретных реализаций сервисов.

6. **Перезапуск или завершение**: по завершении миссии, возникновении критической ошибки или по желанию оператора система может перейти в SAFE_MODE, перезагрузить BIOS или завершить работу. Благодаря модульности, перезапуск одного сервиса не требует остановки остальных.

Заключение

Проект **QIKI OS и ORION** — это гибрид хардкорного симулятора и виртуальной операционной системы. Он объединяет реалистичные модели оборудования, сложную логику агента и удобный интерфейс оператора. Анализ репозитория показал, что команда активно ведёт разработку: ведутся worklog-ы и TODO-файлы для ORION, реализуются модули безопасного режима, инциденты, улучшение инспектора и поддержка plug-and-play.

Сильные стороны: модульность, распределённость, стандартизованные интерфейсы, plug-and-play для устройств, чёткая документация API и событий, удобный TUI для оператора.

Слабые стороны: добавилась сложность оркестрации микросервисов; потребность в поддержке версий API, настройке JetStream и отслеживании состояния сети сообщений. Однако эти недостатки компенсируются гибкостью, тестируемостью и возможностью поэтапного наращивания функциональности.

Проект продолжит развиваться: планируется расширение нейронного ядра (подключение OpenAI для генерации предложений), реализация новых типов миссий, улучшение UX-интерфейса и интеграция с реальными датчиками и исполнительными механизмами.

---

1 ARCHITECTURE.md
https://github.com/sonra44/QIKI_DTMP/blob/7637b6824263dd9bed7eda2e8454121f848064b3/docs/ARCHITECTURE.md

2 3 docker-compose.operator.yml
https://github.com/sonra44/QIKI_DTMP/blob/7637b6824263dd9bed7eda2e8454121f848064b3/docker-compose.operator.yml

4 neuro_hybrid_core_design.md
https://github.com/sonra44/QIKI_DTMP/blob/7637b6824263dd9bed7eda2e8454121f848064b3/docs/design/q-core-agent/neuro_hybrid_core_design.md