ChatGPT

Техническая шаблон-схема для интеграции агента и симулятора QIKI

В этом документе представлена подробная техническая схема (шаблон) для внедрения и развития цифрового двойника QIKI. Схема опирается на текущие конфигурации проекта, на философию сенсоров/исполнительных устройств серии Evochron и на модульную микросервисную архитектуру, описанную в README проекта [1].

1 Обзор архитектуры

1.1 Декомпозиция микросервисов

- **Сервис агента (`q_core_agent`)** – выполняет всю логику принятия решений. Он считывает данные от сенсоров, хранит снимки конечных автоматов (FSM) для каждой подсистемы, генерирует предложения (proposals) через rule- и neural-движки, арбитражирует конфликты и отправляет команды исполнительным механизмам в симулятор. Агент может запускаться в разных режимах: с мок-данными, через gRPC-соединение или с асинхронным хранилищем состояния (state store), что позволяет отделять логику от источников данных.
- **Сервис симулятора (`q_sim_service`)** – эмулирует физический мир. Он поддерживает **модель мира** (координаты, скорость, курс, уровень энергии, статус щитов и т. д.), применяет команды актюаторов и генерирует данные сенсоров. Сервер gRPC предоставляет методы: `GetSensorData`, `SendActuatorCommand` и `HealthCheck`. Симулятор можно заменить или расширить без изменений в агенте [1].
- **Мост FastStream** – дополнительный слой асинхронной передачи событий через NATS. Он транслирует сообщения (например, обновления сенсоров, предложения, данные о состоянии) между агентом и симулятором. Каждое сообщение валидируется моделями Pydantic в `shared/models` и конвертируется в protobuf при необходимости.
- **Регистратор (Registrar)** – ведёт аппенд-лог (append-only), фиксируя каждый сенсорный кадр, команду, переход состояния или ошибку. Он назначает код события (1xx – информационное, 2xx – успешное действие, 3xx – выключение, 4xx – пользовательская ошибка, 5xx – отказ подсистемы, 9xx – авария) для отслеживания и аудита.

1.2 Контракты связи

- **Протоколы (Protocol Buffers)** – определяют типы всех сообщений, передаваемых по gRPC: данные сенсоров (`sensor_raw_in.proto`), команды актюаторов (`actuator_raw_out.proto`), статус BIOS, снимки FSM, предложения (proposals) и API симулятора (`q_sim_api.proto`). Эти файлы являются единственным источником истины для схемы сообщений.
- **Модели Pydantic** в `shared/models` определяют структуру сообщений, передаваемых по FastStream. Они обеспечивают строгую типизацию, валидируются автоматически и преобразуются в/из protobuf.

2 Шаблоны конфигурации

2.1 Конфигурация профиля

Каждый файл профиля (например, `bot_config.json` для наземного робота и `ship_config.json` для космического аппарата) определяет перечень железа и параметры симуляции. Профиль должен включать:

- **Propulsion** – список двигателей или колесных моторов, для каждого указан `id`, `type`, ориентация и статус (включен/ошибочный). В космическом профиле это `ion_drive_array` и четыре RCS-двигателя (`rcs_forward`, `rcs_aft`, `rcs_port`, `rcs_starboard`).
- **Sensors** – каждый сенсор имеет `id`, `type`, дальность, сектор обзора, энергопотребление и особенности. Пространственный профиль включает:
- `long_range_radar` (направление/дистанция/качество, без ID/IFF);
- `navigation_computer` (оценка положения, инерциальная навигация);
- `thermal_scanner` (тепловые сигнатуры);
- `quantum_scanner` (экзотические сигнатуры, эксперимент);
- `short_range_tracker` – логический составной сенсор (иммерция, нав-компьютер, LIDAR).
- **Computing** – список вычислительных модулей (`quantum_ai_processor`, классические процессоры, резервные системы), с ролями (`primary`, `backup`, `security`), массой и порогами отказа.
- **Power** – источники (термоядерный реактор, солнечные панели), накопители (квантовая батарея, суперконденсаторы) и преобразователи. Каждая запись описывает ёмкость, выходную мощность, входную мощность и возможность одновременной зарядки/разрядки.
- **Hull & compartments** – структурные элементы для моделирования столкновений и повреждений. Записи задают количество отсеков, объём, redundancy и связь с системой жизнеобеспечения.

Конфигурации также содержат булевы флаги `lifesupport` и `pressurized`, которые включают или отключают целые подсистемы. Новые сенсоры или двигатели добавляются, определив их `id`, `type` и параметры; конфиг-лоадер автоматически сопоставит их агенту и симулятору.

2.2 YAML-спецификация бота

BotSpec (см. отчёт) – это центральный документ, который описывает компоненты на абстрактном уровне: тип, экспортируемые порты (`provides`), потребляемые порты (`consumes`), дочерние узлы и контракты. Спецификация служит контрактом между дизайнером и разработчиком: симулятор использует её для создания объектов мира, агент – для инициализации модулей и протоколов, а тестовые среды – для проверки соответствия.

Основные секции:

- **Power** – описывает накопители, источники и логику распределения энергии. Важные порты: `dc_out` (магистраль), `load_request` (запросы мощности) и `energy_status` (уровень заряда, состояние).

- **Propulsion** – группирует главный двигатель и маневровые RCS. Интерфейс различает режимы `RAW` (импульсный контроль) и `IDS` (стабилизация инерции); гарантирует, что действия соответствуют ограничениям безопасности.
- **Sensors** – перечисляет все сенсорные устройства с флагами `id_iff` (false для дальнего радара) и областью покрытия. Разворачиваемые сенсоры (`probe`, `station`) определены как подузлы со своим API.
- **Comms** – включает маяк IFF, передатчик `ping` (активный импульс, раскрывает позицию) и основной канал данных.
- **Shields** – описывает распределение энергии по секторам и API смещения/преднастроек (`bias/preset`).
- **Navigation** – включает движок точек назначения (waypoint) с политикой достижения (относительно радиуса радара) и процедурную модель карты для динамической генерации мира.
- **Protocols_pxe** – перечисляет все поддерживаемые протоколы и их жизненный цикл (PRE, ACT, EXIT, ABORT). Протоколы реализуют сценарии вроде уклонения (evasiveness), безопасного сближения, скрытого дрейфа, маскировки сенсоров и полного отключения.
- **Multi-agent layer** – перечисляет всех агентов, шаблон их FSM и переходы. Каждый агент обрабатывает сенсорные данные, формирует предложения и реагирует на результаты.

BotSpec загружается проектом для создания правильных аппаратных абстракций и конфигурации симулятора и агента при старте.

3 Шаблон архитектуры агента

1. **AgentContext** – хранит текущий статус BIOS, снимок состояния FSM и набор кандидатных предложений. Поддерживает асинхронное обновление (при использовании state store) или синхронное (legacy-режим). Реализует функции проверки (например, `is_bios_ok`, `has_valid_proposals`).
2. **Обработчики (Handlers)** – каждый отвечает за одну подсистему:
3. `BiosHandler`: проверяет уровни энергии, жизнеобеспечения, состояние сенсоров. При не критических отказах генерирует коды 4xx, при критических – 5xx.
4. `FsmHandler`: интерпретирует снимок FSM, полученный от симулятора, и обновляет внутреннее представление агента.
5. `ProposalEvaluator`: объединяет предложения rule-движка и нейронного движка, сортирует по полезности/риску/затратам и выбирает оптимальный набор.
6. `RuleEngine`: генерирует детерминированные предложения на основе политики и порогов, определённых в спецификации (например, запуск `EVASIVE_BURN` при коротком времени до столкновения).
7. `NeuralEngine`: опционально создает предложения с помощью машинного обучения; подключаемый модуль, который читает те же данные сенсоров.
8. `BotCore`: абстрагирует отправку команд в симулятор (через gRPC или NATS). Развязывает генерацию команд и транспортный слой.
9. **TickOrchestrator** – управляет главным циклом. На каждом тике считывает данные из провайдера, вызывает обработчики (bios → fsm → proposals → decision) и отправляет принятые команды. Логирует результаты и переключает систему в безопасный режим (`POWERDOWN_FREEZE`) при исключениях.
10. **DataProviders** – общая спецификация источников данных. `MockDataProvider` генерирует входные данные для тестов; `GrpcDataProvider` подключается к удалённому симулятору; `QSimService` использует вызовы в процессе. Провайдер должен поддерживать асинхронный режим для state-store.

11. **StateStore** – опциональное асинхронное хранилище для отделения получения сенсорных данных от цикла агента. Текущий эксперимент реализован на Redis; при миграции необходимо заменить синхронный вызов на асинхронный.

4 Шаблон архитектуры симулятора

1. **WorldModel** – хранит истинное состояние: положение, скорость, курс, заряд батареи, статус щитов, параметры сенсоров и т.п. Предоставляет методы:
2. `update(command)`: применяет команду актюатора, изменяя ускорение или вращение. Должен корректно обрабатывать неподдерживаемые команды и обновлять состояние ошибки.
3. `step(dt)`: интегрирует ускорение в скорость, скорость в положение; обновляет энергию и таймеры деградации сенсоров. Должен соблюдать ограничения (максимальная скорость, безопасный разряд батареи).
4. `get_state()`: возвращает копию состояния для отладки или синхронизации.
5. **Генераторы сенсоров** – для каждого типа сенсора реализуется генератор, который отображает состояние `WorldModel` и окружение в сообщения `SensorReading`. Примеры:
6. `generate_long_range_radar(state, environment)`: возвращает направление, дальность и качество для всех обнаруженных объектов в пределах настроенного диапазона; скрывает ID. Применяет маски теней на основе прямой видимости.
7. `generate_navigation_computer(state)`: возвращает ориентацию/позицию, компенсируя дрейф IMU.
8. `generate_thermal_scan(state)`: возвращает тепловые сигнатуры в поле зрения.
9. `generate_quantum_scan(state)`: возвращает экзотические сигнатуры с вероятностью; можно оставить заглушку.
10. `generate_short_range_tracking(...)`: синтезирует локальное отслеживание целей на основе нав-компьютера, IMU и LIDAR.
11. **Обработчик команд** – `receive_actuator_command` интерпретирует команды от агента. Должен валидировать поля, обновлять модель и корректно обрабатывать неподдерживаемые команды (например, возвращая ошибку).
12. **gRPC-сервер** – реализует методы, определенные в `q_sim_api.proto`. Сервер должен быть без состояния, делегировать вызовы модели мира и генераторам сенсоров, и обеспечивать последовательный доступ к состоянию.

5 Перечень подсистем для космического профиля

Ниже приведён список аппаратных модулей из `ship_config.json` и **BotSpec**. Каждая строка должна соответствовать элементу в кодовой базе или симуляции.

Категория	Идентификатор	Тип	Назначение
Главный движитель	`ion_drive_array`	Ion thruster	Обеспечивает основную поступательную тягу
RCS-двигатели	`rcs_forward`	RCS thruster	Положительное ускорение вдоль X
	`rcs_aft`	RCS thruster	Отрицательное ускорение вдоль X

Категория	Идентификатор	Тип	Назначение
	`rcs_port`	RCS thruster	Положительное ускорение вдоль Y
	`rcs_starboard`	RCS thruster	Отрицательное ускорение вдоль Y
Навигация	`navigation_computer`	Nav computer	Интегрирует IMU и звёздный трекер для инерциальной навигации
Дальний радар	`long_range_radar`	Radar sensor	Обнаруживает направление/дистанцию, без ID/IFF
Ближний трекер	`short_range_tracker`	Tracking sensor	Комбинирует нав-компьютер, IMU и LIDAR для локального трекинга
Детальный скан	`target_scanner`	Detail probe	Возвращает подробную информацию по выбранной цели
Тепловой скан	`thermal_scanner`	Thermal sensor	Обнаруживает тепловые сигнатуры
Квантовый скан	`quantum_scanner`	Quantum sensor	Обнаруживает экзотические сигнатуры (эксперимент)
LIDAR	`lidar_front`	LIDAR sensor	Высокое разрешение дальности на короткой дистанции
IMU	`imu_main`	Inertial unit	Измеряет угловые скорости и ускорения
Секторы щита	`front`, `left`, `right`, `rear`	Shield sector	Поглощают и распределяют входящий урон
Реактор	`fusion_reactor`	Power source	Постоянный источник энергии
Банк батарей	`quantum_battery`	Power storage	Аккумулирует и отдаёт электрическую энергию
Солнечные панели	(неявно)	Energy harvest	Заряжает батареи и капы при освещении
Вычисления	`quantum_ai_core`	AI processor	Выполняет нейронный вывод и высокоуровневое планирование

Категория	Идентификатор	Тип	Назначение
Жизнеобеспечение	`life_support`	Environmental	Управляет атмосферой, теплом, водой (флаг в конфиге)

С развитием проекта таблицу можно расширять новыми сенсорами, движителями и подсистемами. Каждый новый элемент должен быть объявлен в конфигурационном файле, реализован в симуляторе и отображён в агенте.

6 Определения протоколов

Протоколы описывают высокоуровневое поведение и выполняются движком **PXE**. Каждый протокол определён через **предусловия (PRE)**, **действия (ACT)**, **условия завершения (EXIT)** и **условия прерывания (ABORT)**. Ниже приведён шаблон, который можно адаптировать для любого нового протокола:

```
PROTOCOL_NAME:
  preconditions:
    - <условие_1>  # например, hazard_detected, target_verified, sensor_ready
    - <условие_2>
  actions:
    - <команда_актуатора>  # например, set_flight_mode: IDS
    - <команда_щита>       # например, set_shield_bias: toward_threat
    - <команда_сенсора>    # например, enable_masking
  exit:
    - <условие_завершения> # например, risk_grade_decreasing_for_interval
  abort:
    - <условие_прерывания> # например, power_drop
  safety:
    - <правило_безопасности> # например, priority_9xx_preempts
```

Например, `EVASIVE_BURN` использует вышеуказанную структуру с предусловиями типа `hazard_detected` и действиями по перенаправлению и ускорению от угрозы. Определения протоколов следует хранить в отдельном YAML-файле или в коде, чтобы отслеживать версионность.

7 Рекомендации по реализации

- **Валидация**: Реализуйте загрузчик, который читает профиль JSON и YAML BotSpec, проверяя, что все идентификаторы компонентов в профиле присутствуют в спецификации и наоборот.
- **Граф зависимостей**: Постройте граф взаимосвязей «порт поставляет → порт потребляет». Симулятор должен создавать только те узлы, у которых удовлетворены зависимости. Циклы запрещены, кроме как через каналы события.
- **Обработка ошибок**: Все подсистемы должны посылать коды ошибок через регистратор. Каждая подсистема должна поднимать точные коды (4xx, 5xx) и, по желанию, возвращать описательный payload.

- **Тестирование**: Для каждого протокола напишите интеграционные тесты, которые симулируют срабатывание и проверяют команды, состояние сенсоров и записи в регистраторе. Используйте Mock-провайдер для проверки логики агента без симулятора и gRPC-провайдер – для end-to-end.
- **Расширяемость**: Новые сенсоры или двигатели добавляются путем обновления профиля, спецификации и симулятора. Если сенсор возвращает новые поля данных, расширяйте определения протоколов и обновляйте соответствующие агенты. Числовые параметры (диапазоны, пороги) должны храниться в конфигурациях, чтобы инженеры могли настраивать поведение без изменений кода.
- **Соответствие философии Evochron**: Соблюдайте четкое разделение дальнего и ближнего обнаружения, привязывайте достижение точек к текущему радиусу радара, и учитывайте окклюзии как часть модели. Не раскрывайте ID объектов на дальнем радиусе; расширяйте детали только когда цель входит в SR-зону или после явного сканирования. Управление энергией оставляйте простым (несколько режимов) и предпочитайте процедурную генерацию статическим таблицам.

Заключение

Этот шаблон является базой для внедрения и дальнейшего развития платформы QIKI. Он фиксирует архитектуру, определяет повторно используемые шаблоны для сенсоров и актюаторов, и предписывает строгие практики валидации и логирования. Соблюдение этих рекомендаций позволит добавлять новое железо, функции симуляции или AI-поведение, не нарушая надежность и отслеживаемость системы.

---

1 sonra44/NEW
https://github.com/sonra44/NEW%23readme