ChatGPT

План интеграции радара с учётом gRPC, NATS/FastStream и Pydantic v2

Этот документ суммирует весь путь рассуждений в текущей ветке чата и предлагает детальный план по включению радарного сенсора в существующий проект QIKI. В проекте уже используются gRPC, NATS/FastStream и Pydantic v2, поэтому план учитывает их особенности и не ломает действующую архитектуру.

1. Исходное состояние и требования

- **gRPC + NATS/FastStream.** Сервис симуляции (`q_sim_service`) общается с ядром агента (`q_core_agent`) по gRPC, а внутри инфраструктуры появились брокеры NATS и библиотека FastStream для публикации/подписки на события.
  FastStream поддерживает несколько брокеров и обеспечивает проверку и сериализацию сообщений через Pydantic; среди его особенностей — единый API для NATS/Kafka/RabbitMQ, валидация входящих сообщений, автоматическая документация и тестируемость [1] .
- **Pydantic v2.** Проект переходит на Pydantic v2, где настройки модели задаются через `model_config` вместо класса `Config` ; старые декораторы `@validator` и `@root_validator` заменены на `@field_validator` и `@model_validator` соответственно [2] [3] . Это требует обновления моделей данных.
- **Текущие сенсоры.** В `common_types.proto` перечисление `SensorType` пока содержит только LIDAR, IMU, CAMERA, GPS и THERMAL. `SensorReading` может передавать только векторные, скалярные или бинарные данные.
- **Задача.** Добавить поддержку радара, обеспечив вывод таблицы с объектами (позиция, высота, скорость, сигнатуры), понятную как человеку, так и боту. Важно учесть логику транспондера: устройство может передавать свой идентификатор или молчать, но в любом случае остаётся видимым на первичном радаре. Для этого нужно расширить протоколы, симулятор, агент и каналы обмена, не нарушая остальные модули.

2. Исследование стандартов и практик

- **Формат сообщений радара.** В open-source проектах (ROS/Autoware) для одной отметки используются поля **range**, **azimuth**, **elevation**, **doppler velocity** и **amplitude** [4] . Это согласуется с нашей моделью: `range_m` , `bearing_deg` , `elev_deg` , `vr_mps` , `snr_db` / `amplitude` и `rcs_dbsm` . Кроме того, Autoware определяет `RadarTrack` с полями `uuid` , `position` , `velocity` , `size` , `classification` и ковариациями [5] . Сама же `SensorReading` в нашей системе пока не поддерживает массив детекций.
- **Pydantic v2.** Чтобы задать конфигурацию модели, необходимо определить атрибут `model_config` в виде словаря; класс `Config` из v1 считается устаревшим [2] . Для валидации полей теперь нужно использовать `@field_validator` / `@model_validator` [3] .
- **FastStream.** FastStream предоставляет декораторы `@broker.subscriber` и `@broker.publisher` для обработки сообщений. Он автоматически кодирует и декодирует JSON-сообщения, используя Pydantic для валидации. Это упрощает работу с

NATS и позволяет получать типизированные объекты по аннотациям данных [6]. Такой подход подходит для публикации радарных кадров и треков.

- **Транспондер.** В авиации транспондер — часть системы вторичной радиолокации, которая передаёт ответные коды на запросы и значительно улучшает эффективность радаров. Он позволяет диспетчерам быстро идентифицировать самолёты, но его ответы независимы от первичного радарного отражения [7]. Транспондер имеет режимы On/Alt/Standby/Off; даже в режиме Off самолёт отражается на первичном радаре, но у диспетчера нет расширенной информации [8]. Эти идеи можно адаптировать для нашего «IFF» — боты с отключённым транспондером будут видны радару, но будут классифицироваться как неидентифицированные.

3. Дизайн сообщений и протоколов

3.1. Расширение `common_types.proto`

1. **Добавить тип** `RADAR` **в** `SensorType`**.** Это позволит однозначно идентифицировать радарные сообщения.
2. **Определить** `enum Emission` (none, comm, nav, active_radar, thermal, unknown) для описания излучений цели. Эту информацию будет использовать классификатор.
3. **Создать** `message RadarDetection` со следующими полями:
4. `float range_m` — дальность;
5. `float bearing_deg` — азимут;
6. `float elev_deg` — угол места (положительные значения — выше, отрицательные — ниже);
7. `float vr_mps` — радиальная скорость (доплеровский сдвиг);
8. `float snr_db` — отношение сигнал/шум (или амплитуда) [4];
9. `float rcs_dbsm` — эффективная площадь рассеяния (подпись цели);
10. `Emission emission` — тип собственного излучения.
11. **Определить** `message RadarFrame`:
12. `common.UUID sensor_id` — идентификатор сенсора;
13. `repeated RadarDetection detections` — массив отметок;
14. `google.protobuf.Timestamp timestamp` — метка времени;
15. `float signal_strength` — мощность собственного сигнала (можно использовать для регулировки порога детекции);
16. `string source_module` — откуда пришли данные (симулятор, реальный радар и т. д.).
17. **Определить** `enum ObjectType` (unknown, debris, ship, station, drone, asteroid, projectile и т. д.) и `enum FriendFoe` (friend, foe, neutral, unknown). Эти перечисления пригодятся при классификации и строятся на основе ROS/Autoware классификаторов [5].
18. **Определить** `message RadarTrack` (опционально). Это объект-уровневое представление цели после трекинга. Поля: `uuid track_id`, `ObjectType type`, `FriendFoe iff`, `common.Vector3 position`, `common.Vector3 velocity`, `float vr_mps`, `float snr_db`, `float rcs_dbsm`, `Emission emission`, `float quality` (от 0 до 1), `bool transponder_on`. Использование таких треков позволит абонентам получать уже сглаженные данные и реагировать проще.

3.2. Расширение `sensor_raw_in.proto`

- Добавить поле `RadarFrame radar_data` в `oneof sensor_data` структуры `SensorReading`. Тогда `SensorReading.sensor_type` может быть `RADAR`, а данные

будут передаваться в поле `radar_data`. Это сохранило бы обратную совместимость, т.к. старые клиенты игнорируют неизвестные поля.

3.3. Дополнение gRPC API (`q_sim_api.proto`)

- **Расширить** `GetSensorData`: если `sensor_type` равен `RADAR`, сервер возвращает `SensorReading` с заполненным `radar_data`.
- **Добавить метод** `GetRadarFrame` (опционально): `rpc GetRadarFrame(common.Empty) returns (sensors.RadarFrame)`. Это позволит запрашивать кадры без оболочки `SensorReading` и удобно для подписчиков NATS.

4. Модификация симулятора (`q_sim_service`)

1. **Конфигурация.** Вместо одного `sim_sensor_type` добавить список `sim_sensor_types`, где можно указать `RADAR`, `LIDAR`, `IMU` и другие датчики. Для каждого типа можно задавать собственный интервал (`tick_interval`).
2. **Генерация радара.** Реализовать модуль `radar_simulator.py`, который формирует `RadarFrame`. Здесь можно использовать упрощённую физическую модель: несколько объектов (`SimObject`) с координатами и скоростью; добавлять шум на дальность, азимут, высоту и радиальную скорость; порог SNR с CFAR. Классификация (по `rcs_dbsm` / `snr_db` / кинематике) может быть встроена в симулятор для генерации `ObjectType` и `FriendFoe` в `RadarTrack`.
3. **NATS/FastStream издатель.** Создать службу `radar_publisher`, использующую `faststream.nats.NatsBroker`. Она будет подписана на канал внутренних сообщений от симулятора и публиковать кадры или треки в темы NATS. Для примера:

```
from faststream.nats import NatsBroker
from pydantic import BaseModel
from datetime import datetime

class RadarFrameMsg(BaseModel):
    frame: RadarFrame  # импорт из сгенерированных gRPC-моделей

    # Pydantic v2: дополнительные настройки модели
    model_config = {
        'extra': 'forbid',      # запрет лишних полей
        'validate_assignment': True,
    }

broker = NatsBroker()

@broker.publisher('qiki.radar.frames')
async def publish_radar_frame(msg: RadarFrameMsg):
    pass

async def on_new_frame(frame: RadarFrame) -> None:
    await publish_radar_frame(RadarFrameMsg(frame=frame))
```

FastStream автоматически преобразует Pydantic-класс в JSON и отправит его в NATS; обратное преобразование выполняется на стороне потребителя. 4. **gRPC сервис.** В

`QSimService.generate_sensor_data()` добавить ветку для `RADAR`, чтобы при gRPC-запросе возвращать `SensorReading` с `radar_data`. 5. **Совместимость.** После расширения `.proto`-файлов нужно перегенерировать gRPC-модели и обновить импорты. При наличии NATS/FastStream радиоданные могут дублироваться: по gRPC — для ядра агента, по NATS — для остальных сервисов (аналитики, отображения и т. п.).

5. Интеграция в агент (`q_core_agent`)

1. **Расширение моделей.** Добавить Pydantic v2-модели `RadarDetectionModel`, `RadarFrameModel` и `RadarTrackModel`. Эти модели должны использовать атрибут `model_config` для конфигурации и `@field_validator` для проверки диапазонов (например, ограничить `bearing_deg` 0–360°, `elev_deg` –90°...90°, `snr_db` ≥ 0 и т.п.).
2. **Поддержка gRPC.** В `GrpcDataProvider.get_sensor_data()` обработать `sensor_type=RADAR` и вернуть объект `RadarFrameModel`. В функции трекинга/классификации использовать массив `detections` для ассоциации с существующими треками (например, фильтр Калмана). Классификацию `ObjectType` и `FriendFoe` проводить на основании RCS, SNR, к/у и транспондерного сигнала.
3. **Поддержка NATS/FastStream.** Создать `radar_subscriber` для подписки на тему `qiki.radar.frames` и преобразования входящих JSON-сообщений в Pydantic-модели. FastStream обеспечивает автоматическое преобразование сообщений в указанный Pydantic-класс по аннотации, поэтому достаточно объявить функцию:

```
@broker.subscriber('qiki.radar.frames')
async def handle_radar_frame(msg: RadarFrameMsg):
    # msg.frame — Pydantic-модель RadarFrame
    process_frame(msg.frame)
```

Здесь `process_frame()` объединяет детекции в треки, обновляет таблицу и выдаёт предупреждения. 4. **Транспондер и IFF.** При обработке `RadarFrame` из каждого `RadarDetection`/`RadarTrack` извлекается поле `transponder_on` (true/false) и `FriendFoe`. Если транспондер включен, классификатор может определить дружественный/враждебный статус сразу. Если выключен, цель классифицируется как `unknown` до выяснения по другим признакам (траектории, RCS). Боты могут включать/выключать транспондер, что влияет на тактическую неопределённость, но их отметки будут отображаться в таблице всегда. Это соответствует авиадиспетчерской практике: ответы транспондера значительно усиливают отражение и облегчают идентификацию, но их отсутствие не делает самолёт невидимым — отражение от первичного радара остаётся [7]. 5. **Представление в интерфейсе.** Таблица, которую видит пользователь, должна содержать колонки `ID`, `ObjectType`, `Range`, `Bearing`, `Elevation`, `Height`, `RadialSpeed`, `SNR`, `RCS`, `IFF`, `Transponder`, `Note`. Знак плюса/минуса у `Elevation`/`Height` показывает, выше или ниже ли объект. Статус транспондера может отображаться как `ON`/`OFF`/`SBY`.

6. Логика классификации и алертов

1. **Классификатор типа объекта.** На основании эффективной площади рассеяния, соотношения сигнал/шум, скорости и манёвров можно выделять корабли, станции, астероиды, мусор, дроны и т. п. Вероятность принадлежности может храниться в поле `quality` трека.

2. **Определение Friend/Foe.** Если транспондер включён и передаёт код дружественного корабля, трек помечается как `friend`. Если идентификатор совпадает со списком противников — `foe`. При выключенном транспондере статус остаётся `unknown` до проведения дополнительных проверок.
3. **Алгоритм алертов.** Сервис трекинга должен вычислять прогноз времени сближения (range / -radial_speed) и выдавать предупреждения, если объект приближается слишком быстро или находится в опасной зоне. При включённом транспондере можно использовать дополнительные данные (например, запрошенный манёвр). Система алертов может публиковать сообщения в отдельную тему NATS (`qiki.alerts`) с Pydantic-моделью `Alert`, чтобы другие сервисы (интерфейс, ИИ) реагировали.

7. Тестирование и валидация

1. **Перегенерация сообщений.** После изменения `.proto`-файлов нужно перегенерировать Python-клиент и сервер (gRPC), а также Pydantic-модели.
2. **Проверка совместимости.** Убедиться, что старые клиенты (без радара) продолжают работать: `SensorReading` с новым полем `radar_data` должен игнорироваться библиотекой gRPC и Pydantic там, где он не ожидается.
3. **Модульные тесты.**
4. Проверить корректность сериализации/десериализации `RadarFrame` и `RadarTrack` через gRPC и NATS/FastStream.
5. Написать тесты на валидаторы Pydantic v2 (например, `bearing_deg` в диапазоне 0–360°, `elev_deg` в –90°...90°).
6. Смоделировать ситуацию, когда бот выключает транспондер, и убедиться, что его трек всё равно отображается, но помечается как `unknown`.
7. **Интеграционные тесты.** Запуск `q_sim_service`, `radar_publisher`, `q_core_agent` и подписчиков NATS в Docker-compose; проверка передачи сигналов по gRPC и NATS. FastStream поддерживает in-memory тесты, что позволяет писать CI-тесты без настоящего брокера [9] .
8. **Нагрузочное тестирование.** На вашем VPS с четырьмя ядрами и 16 ГБ RAM стоит проверить, сколько объектов/детекций в секунду система выдерживает. Логику трекинга можно распараллеливать, а FastStream/NATS хорошо масштабируются.

8. Причинно-следственная логика и почему так

- **Многоканальность (gRPC + NATS).** gRPC остаётся основным каналом между симулятором и ядром агента: он синхронный, строго типизированный и управляет запросами/ответами. NATS через FastStream — асинхронный канал, предназначенный для широковещательных данных (радар, алерты, логи), где подписчики могут подключаться и отключаться. Разделение каналов упрощает масштабирование: например, аналитика может подписаться только на тему `qiki.radar.tracks` и не мешать основному циклу агента.
- **Pydantic v2.** Использование `model_config` и `@field_validator` делает модели более явными и позволяет управлять поведением (например, запретить лишние поля, автоматически приводить данные к нужному типу). Это обеспечивает безопасность при приёме сообщений с NATS, где потенциально любой сервис может публиковать данные.
- **Классификация и IFF.** Чёткие перечисления `ObjectType` и `FriendFoe` позволяют как ботам, так и людям интерпретировать таблицу одинаково. Без этого нейронный движок или правило может использовать устаревшие строки, а пользователь будет видеть непонятные сокращения.

- **Логика транспондера.** Следуя авиационной практике, транспондеры значительно улучшают идентификацию, но не являются условием видимости объекта. Поэтому модель хранит `transponder_on` отдельно от IFF. Это позволяет игрокам скрываться, но оставаться видимыми; даёт тактическое пространство для обмана (например, включить дружественный код вражескому дрону) и усложняет работу классификатора.
- **RadarTrack как отдельный тип.** Передавая объекты уже после трекинга, можно уменьшить поток данных в NATS и снизить нагрузку на клиентов. В то же время сырые кадры (`RadarFrame`) могут быть полезны для отладки, записи или обучения нейронных сетей. Поэтому обе структуры предусмотрены.
- **FastStream.** Выбор FastStream оправдан: он автоматически проверяет сообщения через Pydantic, поддерживает зависимостьную инъекцию и документирует API (AsyncAPI), что упрощает разработку и снижает вероятность ошибок при отправке/приёме сообщений [6].

9. Заключение

Предложенный план подробно охватывает все аспекты интеграции радара в проект. Он основан на анализе существующей архитектуры (gRPC, NATS, FastStream, Pydantic v2) и на исследованиях стандартов радарных сообщений и транспондеров. Предлагаемые изменения сохраняют обратную совместимость, разделяют синхронные и асинхронные каналы, обеспечивают валидацию и понятность данных как человеку, так и боту. Следуя этому плану, можно реализовать радарный сенсор, интегрировать его в симулятор, агент и систему сообщений, добавить классификацию, логику транспондера и алерты, а также проверить корректность работы через модульные и интеграционные тесты.

---

[1] [6] [9] Streamlining Asynchronous Services with FastStream | NATS blog
https://nats.io/blog/nats-supported-by-faststream/

[2] [3] Migration Guide - Pydantic
https://docs.pydantic.dev/latest/migration/

[4] [5] Data message for radars - Autoware Documentation
https://autowarefoundation.github.io/autoware-documentation/main/design/autoware-architecture/sensing/data-types/radar-data/reference-implementations/data-message/

[7] Surveillance Systems
https://www.faa.gov/air_traffic/publications/atpubs/aim_html/chap4_section_5.html

[8] Different transponder types and their use in aviation
https://www.aerotime.aero/articles/the-different-types-of-transponders-used-in-aviation