Стандарт компонентной и плагинной разработки Metabot v1.0

Инженерные принципы разработки плагинов и модулей Metabot

Зачем это нужно

Metabot развивается как платформа, а не как набор разрозненных скриптов, временных обходов и локальных helper-классов. Поэтому для нас критично не просто “написать рабочий код”, а делать это так, чтобы решения можно было повторно использовать, безопасно развивать, тестировать, документировать и передавать между разработчиками без потери смысла.

Этот стандарт нужен, чтобы:

- уменьшать архитектурный шум и техдолг;
- не превращать платформу в свалку случайных утилит;
- проводить границу между локальным решением и системным компонентом;
- не смешивать бизнес-логику, интеграции, канальные адаптеры и инфраструктурные детали;
- делать новые возможности платформы предсказуемыми для команды, сценаристов и интеграторов.

Главная идея простая: **мы не пишем код “под случай”, мы строим расширяемую инженерную среду**.

---

1. Сначала короткий spec, потом код

Принцип

Любая новая платформенная доработка начинается с короткого спека: контекст, цель, границы изменения, ограничения, критерии приёмки.

Почему это важно

Код без предварительной формулировки задачи почти всегда решает не ту проблему, которую кажется, что решает. Спек нужен не ради бюрократии, а ради того, чтобы команда одинаково понимала:

- что именно меняется;
- где граница ответственности;
- что нельзя сломать;
- как проверяется результат.

Плохо

“Надо быстро подправить два места, чтобы заработало.”

Хорошо

“Нужно ввести общий outbound HTTP-примитив с proxy/retry и перевести на него конкретные точки, не ломая старые одноаргументные вызовы.”

Пример

Если появляется системный модуль для исходящих HTTP-запросов, он должен начинаться не с куска кода, а со спека: где он используется, что считается успехом, какие ошибки

retryable, как он доставляется в V8 и как тестируется.

---

2. Один модуль — одна причина к изменению

Принцип

У класса, сервиса или плагина должна быть одна понятная причина измениться.

Почему это важно

Если модуль одновременно:

- делает HTTP-запросы,
- конвертирует CSV,
- сохраняет файлы,
- определяет MIME,
- проверяет политику безопасности,
- генерирует временные имена,

то он уже не является компонентом. Это комбайн. Такие модули плохо тестируются, плохо переиспользуются и становятся центром роста техдолга.

Плохо

Один `Request`-класс, который “умеет всё”.

Хорошо

Отдельные компоненты:

- outbound HTTP;
- file transfer;
- CSV utilities;
- event adapters;
- policy/context validation.

Пример

Компонент голосового ввода должен отвечать за голосовой ввод, а не одновременно за Telegram payload, загрузку аудио, транскрибацию, файловое хранилище и обработку бизнес-сценария.

---

3. У каждого компонента должен быть явный контракт

Принцип

У компонента должны быть:

- понятные входы;
- понятные выходы;
- предсказуемое поведение;
- задокументированная семантика ошибок и ограничений.

Почему это важно

Компонент без контракта быстро превращается в “магическую штуку”, которую кто-то когда-то написал, а остальные боятся трогать. Это ломает повторное использование и делает развитие платформы случайным.

Плохо

Метод с названием `getFileInfoByUrl()`, который в одних случаях только возвращает метаданные, а в других ещё скачивает файл, создаёт temp file и вычисляет MIME по содержимому.

Хорошо

- `fetchUrlContents()` — получает контент;
- `getFileInfoByUrl()` — получает метаданные;
- `downloadFileFromUrlToBusiness()` — скачивает и сохраняет файл в хранилище.

Пример

Событие `ticket_status_changed` должно явно документировать доступные поля, а не оставлять разработчика угадывать, есть ли там предыдущий статус, текущий статус или только `ticket_id`.

---

4. Неявная передача данных запрещена

Принцип

Компоненты не должны зависеть от скрытых связей, случайных значений в памяти, жёстко заданных путей, доменов, записей в БД или “магии рантайма”, если это явно не является частью контракта.

Почему это важно

Скрытые зависимости делают код хрупким. Он работает только в головах авторов и в конкретном окружении, но не становится частью платформы.

Плохо

- жёсткий путь к папке на диске;
- жёстко заданный домен в коде;
- предположение, что JS-обёртка “как-нибудь увидит” PHP-модуль;
- использование значения, которое другой скрипт когда-то где-то записал.

Хорошо

- пути и домены берутся из конфигурации;
- зависимости передаются явно;
- механизм загрузки модулей описан и воспроизводим;
- мост между PHP, JS и V8 является частью documented delivery model.

Пример

Платформенный модуль не должен знать, что файлы конкретного бота лежат в конкретной директории конкретного сервера. Это не обязанность модуля.

---

5. Канальная логика должна быть спрятана внутри компонента

Принцип

Сценарии и верхнеуровневая логика должны описывать **что происходит**, а не вручную реализовывать Telegram, Max, webhook-переходы, форматы multipart или повторные попытки запросов.

Почему это важно

Когда канальная и инфраструктурная логика просачивается наружу, платформа перестаёт быть платформой. Она превращается в набор сценариев с вшитой механикой конкретных каналов.

Плохо

Сценарий сам знает, как устроен Telegram file API, как формируется URL, как грузится файл через прокси и как повторять запрос.

Хорошо

Сценарий обращается к компоненту:

- `VoiceInput`
- `ProxyFetch`
- `ChannelArtifactResolver`

- `TicketStatusChangeEvent`

Пример

Если платформа начинает работать и с Telegram, и с Max, и с другими каналами, различия между ними должны жить в адаптерах и компонентах, а не размазываться по V8-скриптам.

---

6. Общий платформенный код не должен зависеть от конкретного проекта

Принцип

Если код претендует на статус общего системного компонента, он должен быть переносимым и пригодным для повторного использования.

Почему это важно

Компонент, который работает только в одном проекте, на одном домене, в одном окружении или с одним bot ID, не является компонентом платформы. Это проектный helper.

Плохо

- жёстко заданные пути;
- жёстко заданные URL;
- привязка к конкретному боту, инстансу, бизнесу или окружению;
- хранение инфраструктурной информации внутри логики модуля.

Хорошо

- конфигурация выносится наружу;
- файловые пути вычисляются через storage layer;

- URL строятся через общие сервисы;
- компонент можно использовать повторно без переписывания.

Пример

Общий HTTP-модуль — хороший кандидат на платформенный слой. Утилита, которая пишет файлы только в конкретную папку конкретного инстанса, — нет.

---

7. Legacy не расширяем дальше как новый стандарт

Принцип

Если старый модуль уже используется во многих местах, это не значит, что его нужно продолжать развивать как основной архитектурный центр.

Почему это важно

Legacy-код может быть полезен как compatibility layer, но если в него продолжать складывать новые обязанности, он становится точкой системного разложения.

Плохо

Ради новой задачи расширять старый helper ещё сильнее, потому что “он и так уже везде используется”.

Хорошо

- новый системный слой вводится отдельно;
- старый модуль признаётся legacy;
- миграция идёт постепенно;
- новые доработки делаются на новом стандарте.

Пример

Если есть старый модуль, который уже умеет и HTTP, и файлы, и CSV, и multipart, это не повод использовать его как базу для новой платформенной логики. Это повод остановить его рост и начать выносить отдельные системные примитивы.

---

8. Сначала системный примитив, потом точечная миграция

Принцип

Повторяющиеся инженерные проблемы должны решаться один раз как системный примитив, а не много раз в разных helper-классах.

Почему это важно

Когда прокси, retry, таймауты, fallback-логика и диагностика реализуются в разных местах по-разному, это не ускорение, а размножение будущих багов.

Плохо

- отдельно прокси для Telegram;
- отдельно retry для файлов;
- отдельно download helper в интеграционном плагине;
- отдельно ещё один wrapper “на всякий случай”.

Хорошо

- единый outbound HTTP-слой;
- единые правила proxy/retry/timeout;
- поверх него — разные политики:
  - generic fetch,
  - file info,
  - strict file download.

Пример

Если проблема в `file_get_contents` без прокси возникает в нескольких местах, правильное решение — не “пропатчить ещё одну точку”, а ввести платформенный способ исходящих HTTP-запросов и постепенно перевести туда нужные вызовы.

---

9. Границы контекстов должны быть названы

Принцип

У каждого модуля и сервиса должен быть понятный контекст: о чём он и о чём он не должен знать.

Почему это важно

Без границ всё начинает прилипать ко всему. HTTP начинает знать про файлы, файлы — про каналы, каналы — про бизнес-правила, а бизнес-правила — про способы доставки JS-модулей.

Плохо

Один модуль сразу “про интеграции”, “про файлы”, “про экспорт”, “про CSV”, “про загрузку”, “про политики” и “про события”.

Хорошо

Отдельные контексты:

- Outbound HTTP
- File Transfer
- Event Contracts
- CSV/Artifacts
- Channel Adapters
- Runtime Delivery

Пример

`ticket_status_changed` — это контекст событий и контрактов. `ProxyFetch` — это контекст исходящего HTTP. `downloadFileFromUrlToBusiness()` — это контекст скачивания и сохранения файла. Смешивать это в одну сущность нельзя.

---

10. Любая платформенная фича обязана приехать вместе с четырьмя хвостами

Принцип

Код без доставки и сопровождения не считается завершённой фичей.

Минимальный комплект

- spec;
- changelog;
- docs;
- тестовый сценарий.

Почему это важно

Если код есть в git, но:

- runtime его не видит;
- V8 не находит модуль;
- никто не знает, как им пользоваться;
- он не отражён в версии;

то это не готовая функциональность, а полуфабрикат.

Пример

Новый системный модуль должен:

- быть описан в `specs`;
- попасть в changelog версии;
- иметь документацию по подключению и использованию;
- быть реально проверен на stage.

---

11. Перед началом реализации команда должна ответить на пять вопросов

Обязательные вопросы

1. Какова цель изменения?
2. Где проходит граница модуля или сервиса?
3. Какие контракты меняются?
4. Какие инварианты нельзя нарушить?
5. Как проверяется результат?

Почему это важно

Если на эти вопросы нет ответа, значит команда ещё не проектирует систему, а только реагирует на симптомы.

Пример

Если в событие смены статуса добавляется `previous_status_id`, это не “маленькая доработка”. Это изменение event contract. Значит, нужно заранее понять:

- кого оно затронет;
- как это будет документировано;
- что останется backward-compatible;
- как будет выглядеть payload события после изменения.

Практические антипаттерны

Антипаттерн: “универсальный helper”

Признаки

- делает слишком много;
- его имя уже не соответствует содержанию;
- в него продолжают складывать новые обязанности;
- его боятся трогать, но продолжают использовать.

Что делать

- перестать расширять;
- зафиксировать как legacy;
- выделить новые системные примитивы.

Антипаттерн: “разовый фикс становится стандартом”

Признаки

- решение делалось локально;
- потом на него начинают ссылаться как на общий подход;
- контракта и docs нет;
- оно не готово к повторному использованию.

Что делать

- либо честно оставить это локальным кейсом;
- либо поднять до уровня полноценного компонента.

Антипаттерн: “доставка не доведена”

Признаки

- код существует, но среда его не видит;
- модуль доступен только после ручной магии;
- часть живёт в git, часть в БД, часть “должна сама подцепиться”.

Что делать

- определить единый delivery model;
- сделать загрузку модулей воспроизводимой и проверяемой;
- перестать рассчитывать на “оно, наверное, сработает”.

Merge-checklist для платформенного кода

Перед merge каждый новый платформенный модуль должен пройти проверку:

1. Есть ли короткий spec?
2. Понятна ли одна причина к изменению?
3. Есть ли явный контракт входов и выходов?
4. Нет ли скрытых зависимостей или жёсткого хардкода?
5. Не смешаны ли несколько контекстов в одном модуле?
6. Это системный примитив или проектный helper?

7. Не развиваем ли мы legacy-комбайн вместо нового слоя?
8. Есть ли docs, changelog и stage-сценарий проверки?

Итоговая позиция

Платформа Metabot развивается не через случайные удобные утилиты, а через **осмысленные, ограниченные, документированные и повторно используемые компоненты**.

Наша цель — не просто ускорить написание кода, а построить среду, в которой:

- новый разработчик может безопасно продолжить чужую работу;
- сценарии не превращаются в инфраструктурный код;
- интеграции не тащат за собой хаос;
- платформа остаётся расширяемой, а не расползается в стороны.

Каждая новая доработка должна отвечать не только на вопрос “как это сделать”, но и на вопрос “где это должно жить как часть системы”.

Версия #1
Artem Garashko создал 2026-04-02 19:14:26 UTC
Artem Garashko обновил 2026-04-02 19:14:53 UTC