ChatGPT

Эффективное сокращение токенов при анализе Android-проекта моделью Claude

Зачем сокращать токены? При отправке целого Android-проекта на анализ ИИ-модели (например, Claude) легко превысить лимиты контекста и получить огромные счета за использование. Стоимость и время обработки запроса растут линейно с числом входных и выходных токенов [1]. Вместо передачи всего кода целиком, следует оптимизировать контекст – дать модели только необходимую информацию для понимания бага и выработки решения. Ниже приведён краткий гайд по снижению токенов без потери важного контекста.

Отбор необходимых частей проекта

Первый шаг – **изолировать и отправить только те части кода, которые действительно относятся к багу**. Полный проект может включать десятки модулей и тысяч файлов, но модели обычно не требуется видеть их все. Выясните, где именно проявляется баг, и отберите соответствующие фрагменты:

- **Классы и модули, связанные с багом:** Включите исходники только тех компонентов, в которых проявляется ошибка (например, класс экрана, логики или модуля, где происходит сбой). Исключите посторонние модули. Такой подход "выборочно передавать релевантный исходный код" считается наиболее разумным и экономичным [2]. Фактически, идея инструментов вроде 16x Prompt – дать возможность вручную пометить нужные файлы для добавления в prompt [2].
- **Конфигурационные файлы:** Если баг может быть связан с настройками, приложите соответствующие конфиги. Например, проблемы с компонентами или разрешениями в Android часто требуют включить `AndroidManifest.xml` или нужные части Gradle-файлов. Однако не отправляйте *все* конфигурации – только те, что влияют на баг.
- **Логи и описание ошибки:** Если есть сообщение об ошибке, стек-трейс или номер issue, кратко изложите их в запросе. Это поможет модели понять контекст бага без чтения всего проекта. По сообщению об ошибке Claude сможет сузить область поиска проблемы.
- **Исключение нерелевантных частей:** Не включайте ресурсы (изображения, строки локализации, макеты), библиотеки и другие файлы, не влияющие на проблему. Например, если баг в логике Kotlin/Java, XML-разметки UI можно не приводить, **если** они не участвуют в ошибке. Чем "чище" и целевее контекст, тем меньше лишних токенов тратится на данные "для шуму". Практика показывает, что основное преимущество больших окон контекста – вмещать **только нужные модули** проекта [3], а не всё подряд.

Предварительная фильтрация и сжатие данных

После определения круга нужных файлов, **подготовьте эти данные**, отфильтровав и сжав их по возможности. Цель – убрать всё лишнее (комментарии, повторяющийся код, незначимые части), сохранив суть. Вот несколько методов:

Метод	Описание	Польза для сокращения токенов
Структура проекта (`tree`)	Сформируйте дерево каталогов с файлами проекта, чтобы дать модели обзорную структуру без деталей кода [4]. Ограничьте глубину до релевантных директорий. Например, покажите модуль и пару файлов, где проявляется баг.	Обеспечивает контекст о проекте *в краткой форме*. Модель поймёт, как организован код, не читая каждый файл.
Поиск ключевых слов (`grep`)	Прогоните поиск по коду (например, по сообщению об ошибке, названию метода или переменной, связанной с багом). Извлеките несколько строк вокруг найденных совпадений.	Позволяет выделить только *фрагменты* кода, относящиеся к проблеме. Вместо всего файла модель увидит нужные 5–10 строк, экономя контекст.
Сравнение версий (`diff`)	Если известен коммит/изменение, вызвавшее баг, получите `git diff` между рабочей и проблемной версией. Передайте только изменённые строки.	Дифф-патч с конкретными изменениями обычно очень компактен. Claude сфокусируется на отличиях, где, вероятно, и кроется ошибка, без остального кода.
Анализ AST / регулярные выражения	Используйте парсер кода или regexp, чтобы извлечь целевые элементы: например, конкретную функцию или класс целиком, *без* остального кода файла. Также можно автоматически удалить комментарии, отступы и строки, не влияющие на логику.	AST-анализ даст *чистый код* нужного элемента. Удаление комментариев и форматирование снижает количество токенов, не затрагивая смысл программы.
Резюме и сжатие текста	Если файл всё же очень большой, можно сначала подсуммировать его содержание своими словами или с помощью модели, а в prompt дать краткое описание вместо полного текста. Также можно сжать идентификаторы (например, переименовать длинные имена переменных на короткие псевдонимы перед отправкой).	Краткое описание логики или укороченные имена занимают значительно меньше токенов, чем исходный код. Однако важно, чтобы при этом сохранялся контекст, достаточный для поиска решения.

Некоторые из этих техник требуют осторожности. Например, *полное* удаление всех комментариев может лишить модель подсказок (например, TODO или FIXME в коде). Вместо этого можно удалить многострочные описания, но сохранить короткие пометки, касающиеся бага. Ещё одна продвинутая идея – сгенерировать “карту” проекта: список всех функций и классов с их сигнатурами (например, с помощью `ctags`). Сообщество отмечает, что подобная карта (без реализаций, только объявление структур) в начале prompt даёт модели быстрый ориентир по проекту [4]. Это компрессирует код до “содержания” и может облегчить поиск решения.

Важно также **отсечь повторяющийся или стандартный код**. Если баг в пользовательском коде, нет смысла тратить контекст на вставку длинных фрагментов библиотечных функций или

сгенерированного кода, которые заведомо не исправляются. Достаточно указать, какая библиотека или версия используется, вместо включения её исходников.

Структура запроса и сокращение объёма ответа

Помимо сокращения входных данных, правильно сформулированный **промпт** поможет уменьшить количество **выходных** токенов, то есть длину ответа Claude:

- **Укажите требуемый формат ответа.** Чётко дайте понять модели, что вы хотите получить. Например: *«Предоставь исправление в виде diff-патча, без переписывания всего файла»*. Тогда вместо полного листинга кода Claude вернёт только изменения (строки с `-` и `+`), что значительно короче. Новые версии LLM уже сами стремятся давать правки диффом вместо перепечатывания всего файла [5], но явное указание формата повысит шансы сжатого ответа.
- **Просите минимальный необходимый ответ.** Избегайте общих вопросов вроде “как исправить этот баг?”, на которые модель может ответить развёрнутым объяснением. Вместо этого сформулируйте задачу точнее: *«Исправь баг X. В ответе укажи только измененный код и коротко объясни суть исправления»*. Так Claude сфокусируется на патче, а не на длинных пояснениях. Если пояснение нужно, ограничьте его: напр., *«2-3 предложениями опиши причину и подтвердить, что баг решён»*.
- **Исключите дублирование контекста в ответе.** Модель иногда повторяет части вопроса или цитирует куски кода из prompt перед изменением. Можно явно попросить этого не делать: *«Не включай исходный код без изменений в ответе»*. Тогда Claude не станет заново перечислять весь контекст, который вы и так предоставили, а сосредоточится только на новых фрагментах.
- **Примеры и подсказки для краткости.** Если есть возможность, приведите пример, *как должен выглядеть ответ*. Например, можно показать шаблон:

```
```diff
- old line
+ new fixed line
```

Комментарий: Исправил неверную проверку границ массива. ```

(здесь квадратные скобки опущены намеренно для демонстрации)

Такой образец демонстрирует идеальный формат: код-дифф + краткий комментарий. Claude будет стремиться ему следовать. - **Контроль стиля через системные подсказки.** В Anthropic Claude можно использовать системное сообщение (role prompt) или аннотации, чтобы задать стиль ответа. Например: `<responseStyle brevity="high" format="diff+text"/>` (условно) – некая разметка или просто текст *«Ответь максимально кратко, только код и краткий комментарий»*. Это не гарантия, но может повлиять на поведение модели.

Ниже – пример, сочетающий эти приёмы:

```
<системное сообщение>: Ты – помощник по исправлению кода. Строй ответы кратко
и по делу.
```
```

```
<пользователь>:
Проект: Android, Kotlin. Баг: приложение падает при нажатии кнопки "Login".
Причина (вероятно): NullPointerException в UserManager.
Ниже код класса UserManager с пометкой места ошибки.

```kotlin
class UserManager {
    fun login(user: User?) {
        // ...
        if (user.token == "admin") {  // <-- NPE here
            authorize(user)
        }
        // ...
    }
}
```

Задание: Предложи исправление бага. Ответ дай в формате diff-патча (только изменённые строки) с коротким пояснением причины ошибки. ```

В ответ Claude должен выдать только изменённый фрагмент кода (например, добавив проверку `user != null`) и пару предложений комментария. Такой подход исключает долгие рассуждения и пересказ кода, уменьшая объём выдачи.

Потоковая и итеративная подача контекста

Наконец, рассмотрим **итеративный** подход взаимодействия с моделью. Вместо отправки всех данных одним запросом, можно подгружать их частями по мере необходимости:

- **Диалог с уточнениями.** Начните с запроса, описывающего проблему и, возможно, общую структуру проекта, но без деталей кода. Попросите Claude указать, какая информация нужна для решения. Модель может сама запросить конкретные файлы или классы. Затем последовательно предоставьте эти файлы в новых сообщениях. Такой пошаговый диалог экономит токены, потому что на каждом шаге контекст ограничен только тем, что актуально на данный момент. Например, сначала модель может спросить: *«При каком условии возникает NPE? Покажите реализацию функции login.»* – вы отправляете код функции. Потом Claude выдаёт фикс. В результате вы никогда не отправляете лишний код, модель не анализирует то, что ей не понадобилось.
- **Использование инструментов Claude (MCP).** Если вы запускаете Claude в Docker через MCP-сервер, можно задействовать возможности *агента*. Claude способен сам **искать и читать файлы** из подключённого тома, если разрешить ему доступ через MCP [6]. Практически это означает, что вы можете дать модели команду вроде: *«Открой файл UserManager.kt»* – и она сама загрузит содержимое файла в контекст. Модель может по логике задачи определять, какие файлы ей нужны: аналогично тому, как агент в Android Studio на запрос *«Make dark mode default...»* самостоятельно находит соответствующие файлы и вносит правки [7] [8]. Этот подход превращает решение бага в диалог агента с вашим проектом: Claude поочерёдно запрашивает разные части кода, а вы (или окружение MCP) их предоставляете. В итоге **контекст распределяется по нескольким сообщениям**, и ни одно сообщение не перегружено всем проектом сразу.
```

Обратите внимание, что потоковая подача данных требует поддержки со стороны клиента/сервера. В API Anthropic Claude можно реализовать многоходовый обмен сообщениями, поддерживая “память” о прошлом контексте. Также убедитесь, что у Claude есть права на доступ к файловой системе через MCP (в настройках разрешений). Безопасность превыше всего: предоставляя модели доступ к файлам, ограничьте её права только рамками проекта (например, монтируя проект как read-only volume).

Следуя этим рекомендациям, вы сможете значительно **сократить число входных и выходных токенов** при анализе Android-проекта моделью Claude. В итоге взаимодействие станет быстрее, дешевле и эффективнее, а модель по-прежнему получит весь нужный контекст для предложения качественного исправления бага. Помните, что ключ – **дать ровно столько информации, сколько нужно для решения задачи, не больше**. Это искусство сродни хорошей архитектуре кода: ничего лишнего, только необходимое [3].

---

[1] [2] [3] [4] Is it possible to analyze my entire codebase? : r/ChatGPTCoding
https://www.reddit.com/r/ChatGPTCoding/comments/1ba6zot/is_it_possible_to_analyze_my_entire_codebase/

[5] GPT-4.1 глазами веб-разработчика: возможности, интеграция и примеры / Хабр
https://habr.com/ru/articles/900904/

[6] Claude Code Best Practices \ Anthropic
https://www.anthropic.com/engineering/claude-code-best-practices

[7] [8] Agent Mode | Android Studio | Android Developers
https://developer.android.com/studio/gemini/agent-mode