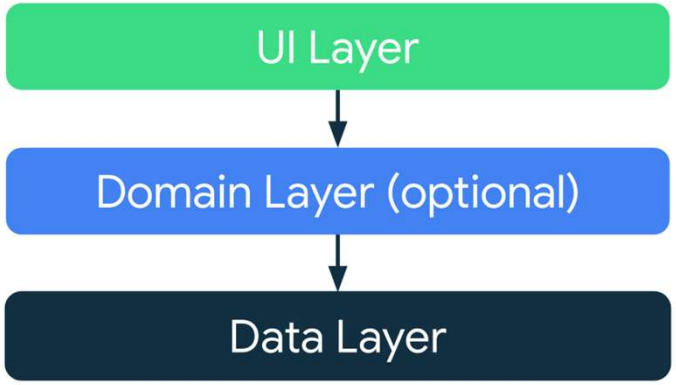
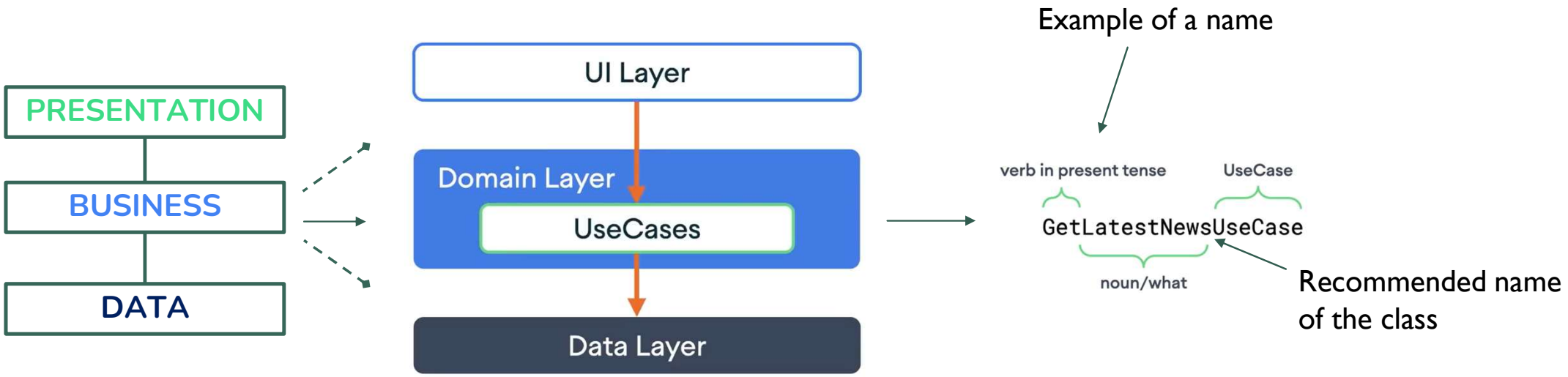
DOMAIN LAYER

INTRODUCTION

- The Domain layer is an *optional* layer that contains the business logic (what to do with business data)
- Business data are not UI related data and represents the 'value' of the application
- For simple app, the business logic can be implemented in the data layer, or even in the ViewModel
- The domain layer increases the testability, modularity, and reusability of the software
- "The domain layer is responsible for encapsulating complex business logic, or simpler business logic that is reused"

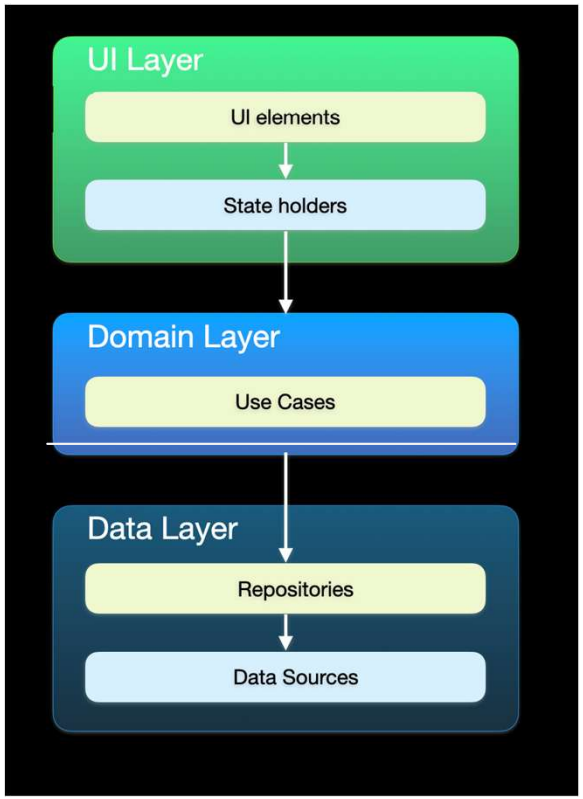


INTRODUCTION



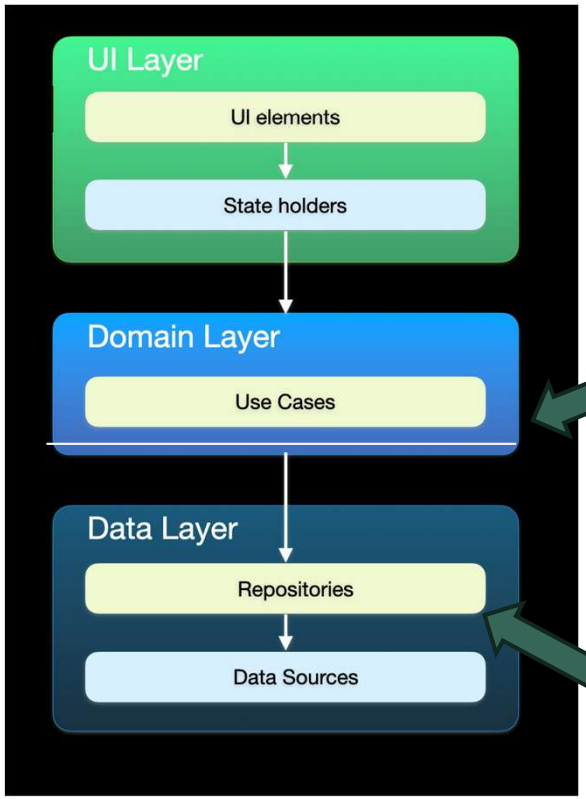
- The components in this layer are just classes, commonly called *use cases* or *interactors*.
- Because they are normal classes, they are not lifecycle aware
- Each class follows a name convention highlighting the use case is responsible over a *single* task.
- A use case uses the data layer, and it is used by the UI layer

DEPENDENCY



- Use case class is stateless
- It may depend on data fetched from the data layer
- The dependency is on the repositories in the data layer
- The domain layer defines an abstract interface to the repositories

DEPENDENCY



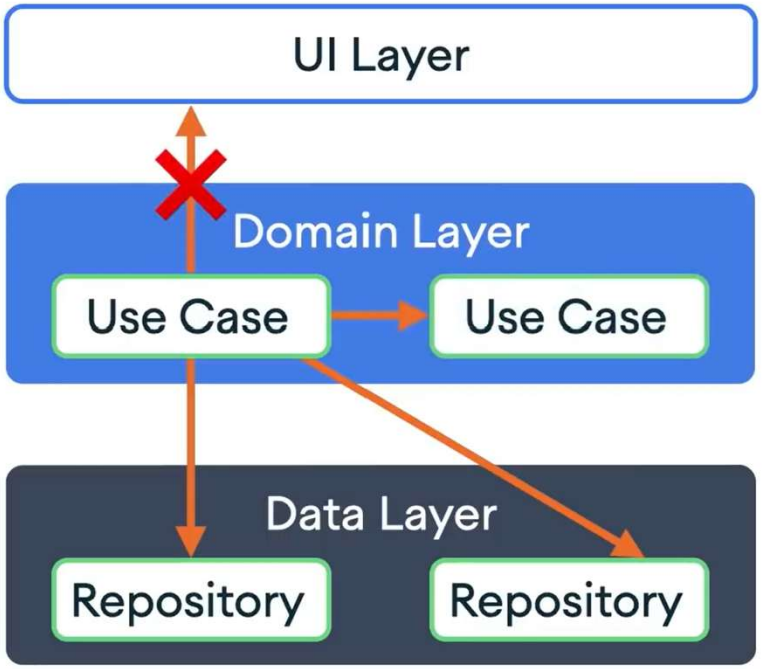
- The domain layer defines the interface to the data
- The useCase uses the operation of the interface

```
interface UserRepository {
suspend fun getUserById(id: String): Result<User>
suspend fun getAllUsers(): Result<List<User>>
suspend fun saveUser(user: User): Result<Unit> }
```

```
// DOMAIN LAYER - Use Case //
class GetUserByIdUseCase( private val repository: UserRepository ) {
    suspend operator fun invoke(userId: String): Result<User> {
 // Business logic here if needed
return repository.getUserById(userId)
}
}
```

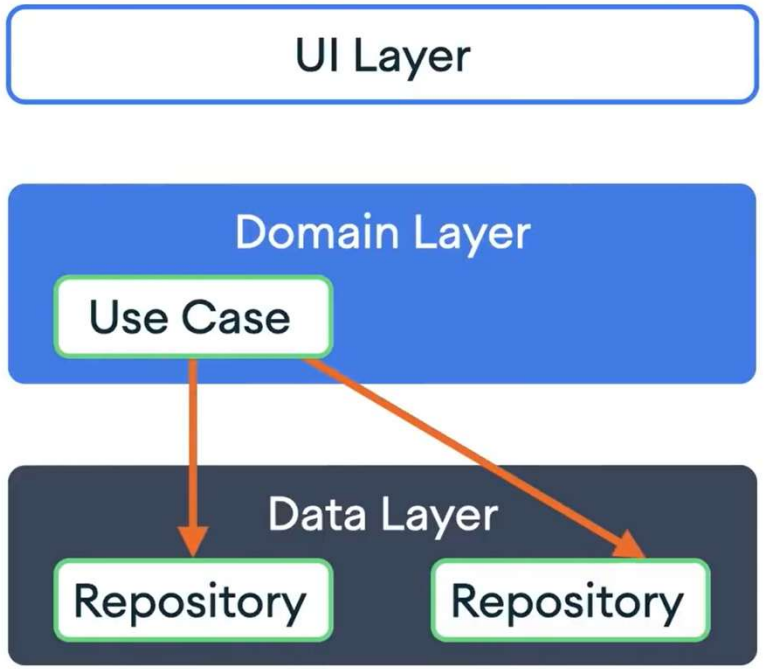
The data layer implements the interface

USE CASE DEPENDENCY



- It can also depend on other uses cases
- It has not to depend on the UI layer
- Recommendation #1: The dependency should by Injected in the class
- Recommendation #2: use case must be stateless (only use immutable data)

EXPOSITION DATA OR OPERATION



- The layer exposes data and operations as flows, StateFlow or suspend function
- A stateFlow is a container of a single data that can be observed
- A Flow is a type that can emit multiple values sequentially over a period of time.

EXAMPLE

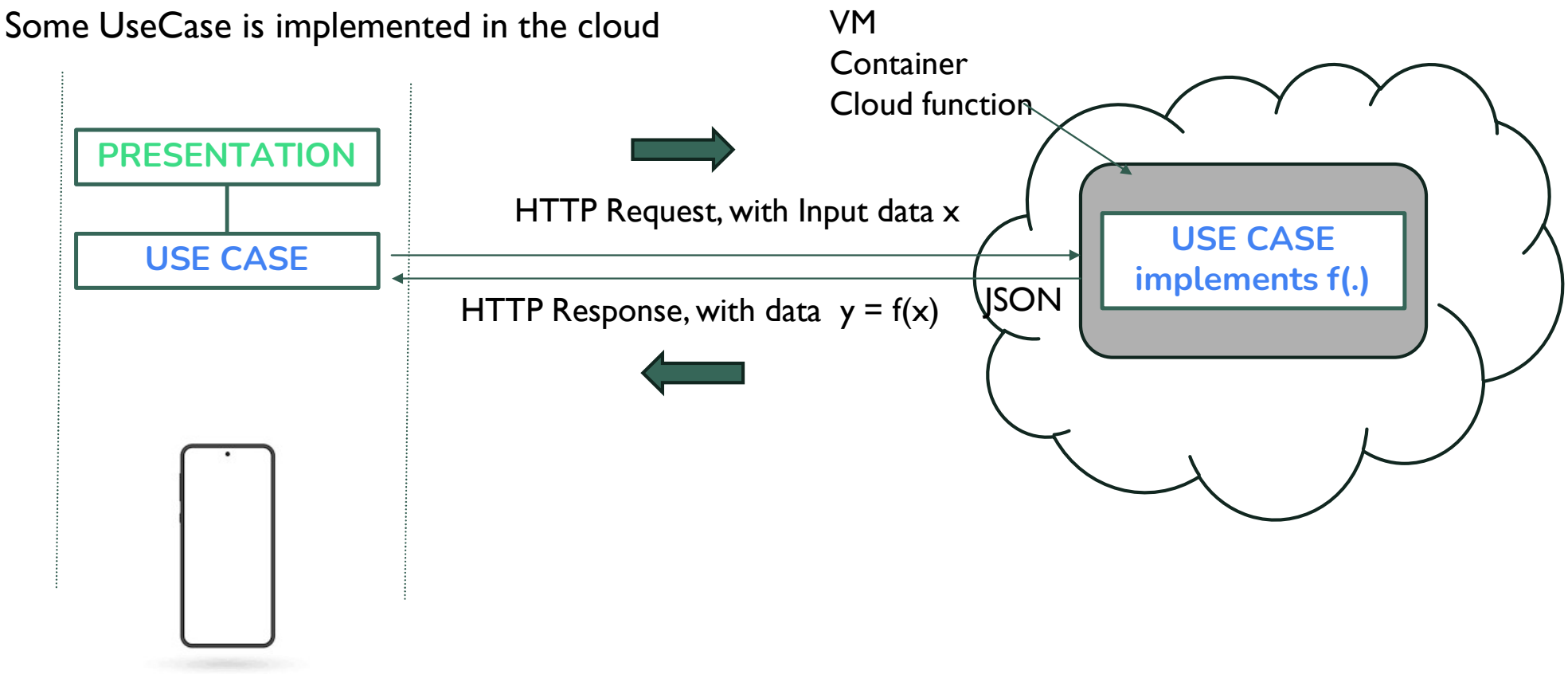
```
class CheckWinnerUseCase {  2 Usages
    operator fun invoke(board: List<Player>, player: Player): Boolean {
        val winningCombos = listOf(
            // Rows
            listOf(0, 1, 2), listOf(3, 4, 5), listOf(6, 7, 8),
            // Columns
            listOf(0, 3, 6), listOf(1, 4, 7), listOf(2, 5, 8),
            // Diagonals
            listOf(0, 4, 8), listOf(2, 4, 6)
        )
        return winningCombos.any { combo ->
            combo.all { index -> board[index] == player }
        }
    }
}
```

- This Use case evaluates he configuration of a board to check if game can continue
- The invoke operator makes the class callable as a function
- The suffix UseCase helps to recognize that it is indeed a class
- The use case exposes one function that has the same name of the class: CheckWinnerUseCase

BUSINESS LOGIC SPLIT

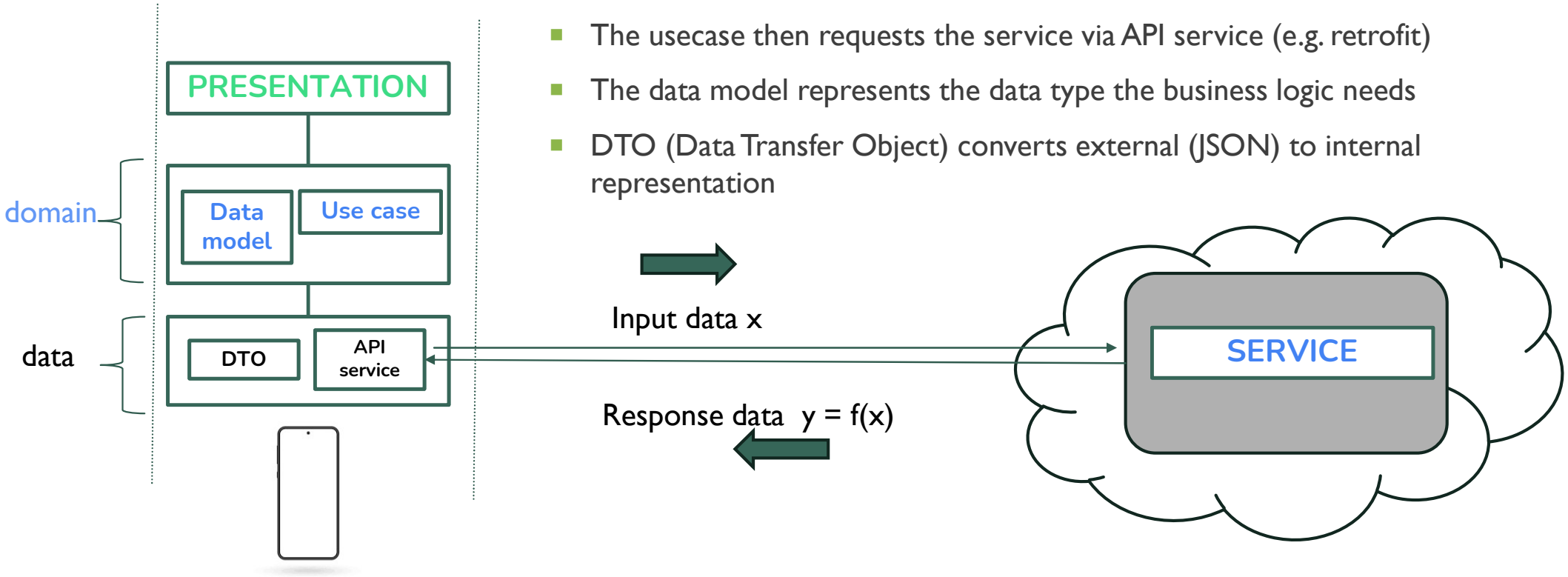
- It is possible to split the business logic in two parts: one running on-board (on the device) and other on the cloud
- We first consider the simpler case where the business logic has no state (no storage), which is referred to as computation offloading pattern
- The need for computation offloading arises for example when complex cpu-consuming algorithms are executed

OFFLOADING PATTERN (COMPUTATION OFFLOAD)



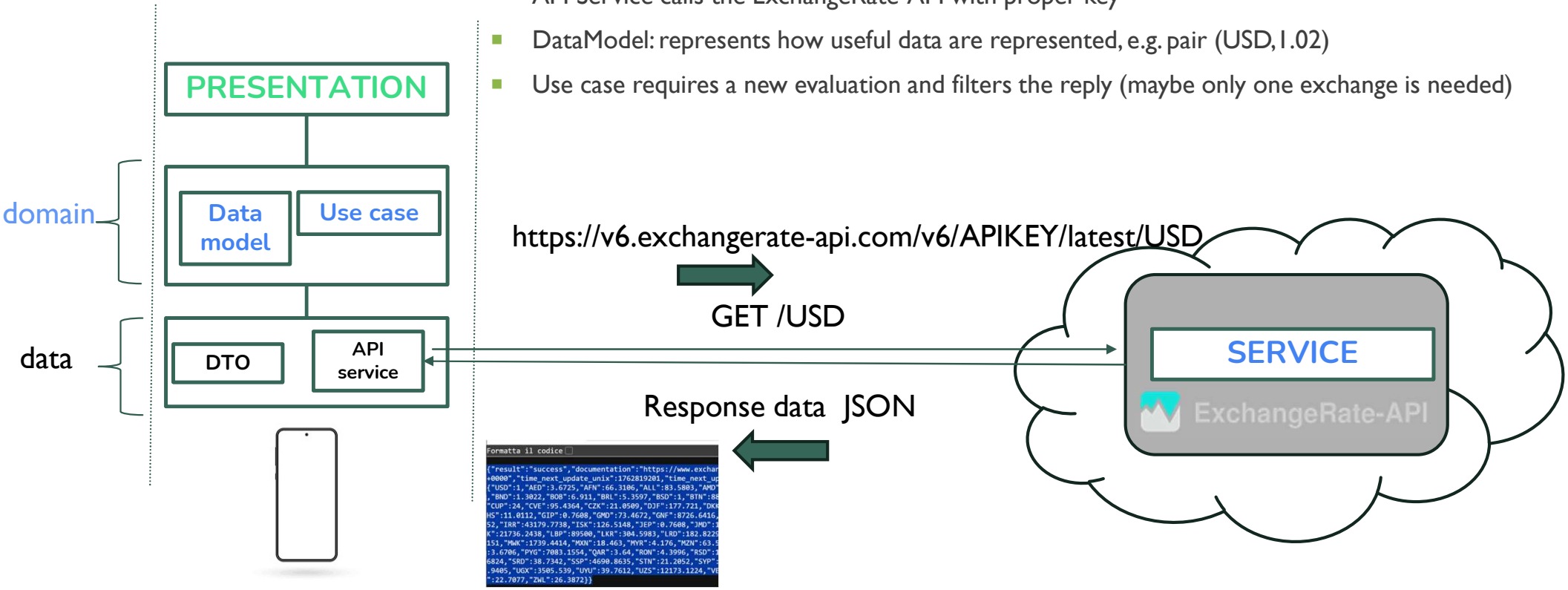
USE ESTERNAL SERVICE

- The business logic can include external services as well
- The usecase then requests the service via API service (e.g. retrofit)
- The data model represents the data type the business logic needs
- DTO (Data Transfer Object) converts external (JSON) to internal representation



EXAMPLE

- DTO: Internal representation of the reply (convert JSON to Kotlin type)
- API Service calls the ExchangeRate-API with proper key
- DataModel: represents how useful data are represented, e.g. pair (USD,1.02)
- Use case requires a new evaluation and filters the reply (maybe only one exchange is needed)



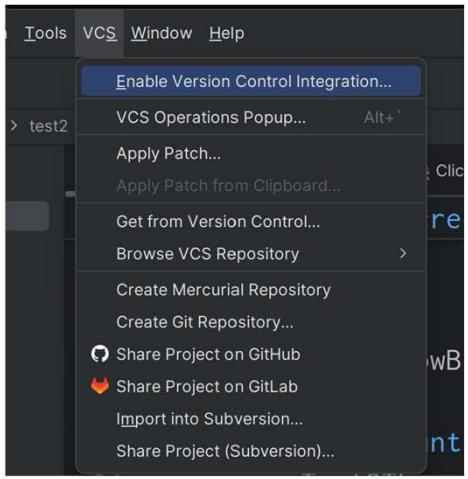
COMPUTATION OFFLOAD

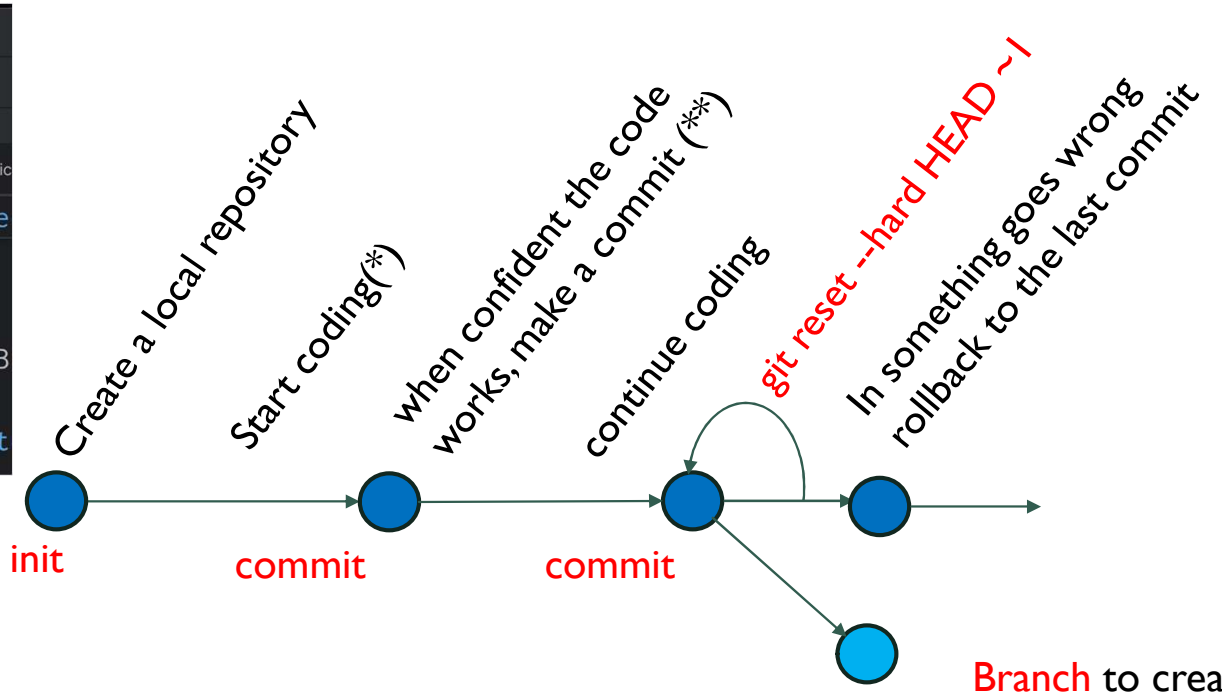
- When part of the business logic is offloaded, function calls are no longer local.
- The semantics of a function call changes as it becomes a remote operation.
- A call changes from synchronous and reliable, to asynchronous and potentially unreliable
- A local call is: fast, reliable, shares memory with other part of the code, exceptions are easy to handle
- remote calls are not 'transparent': they may fail due to network error, latency is non-deterministic, data serialization/deserialization (marshalling/unmarhalling) is require, must handle retries, idempotence, fallbacks and cache
- off-line first design is mandatory
- Some mature external services provide sw libraries to access (like firebase, FB, etc..)

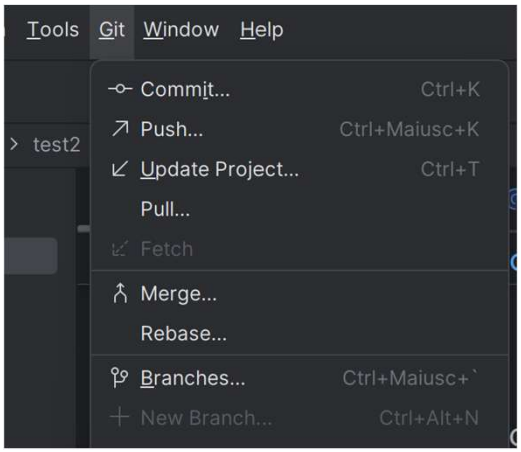
SUGGESTED DEVELOPMENT FLOW (1/2)

- Identify use cases, i.e. the business logic and the related data (often from storyboard)
- Design UI and navigation flow

SUGGESTED DEVELOPMENT FLOW (2/2)



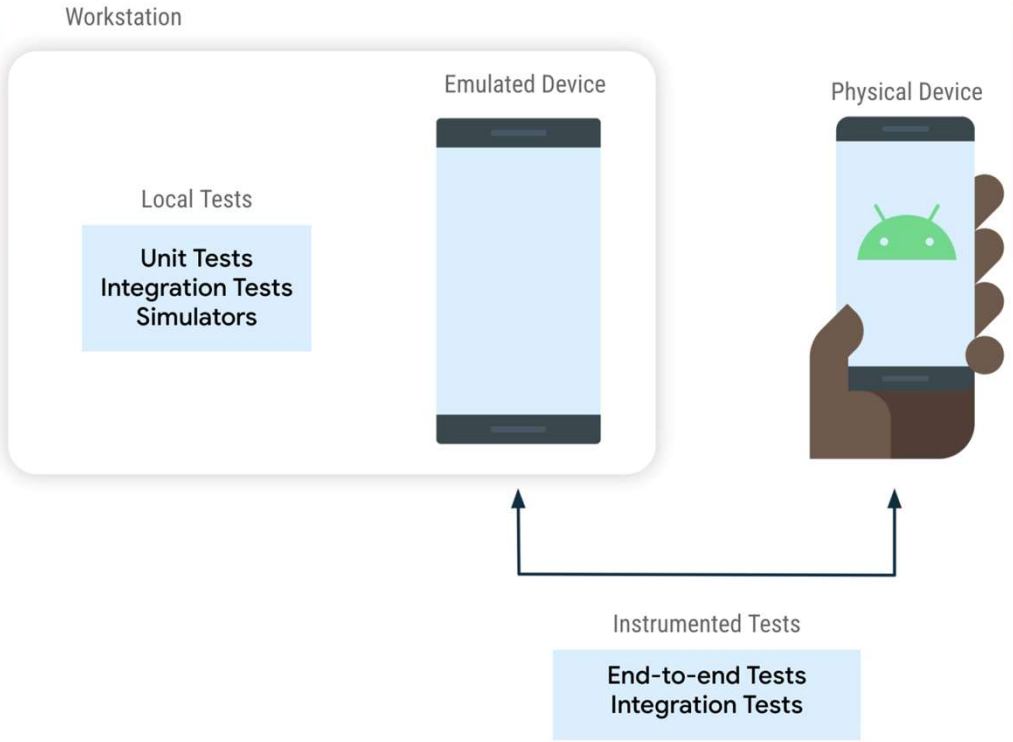




(*)Use the agent-mode to generate code
Use long prompts with detailed technical descriptions
Ask reasonable sized component: (functions, or classes are good candidates)
(**)Generate unit tests and E2E test

Branch to create a new version
Android Studio allows to switch from on version to another

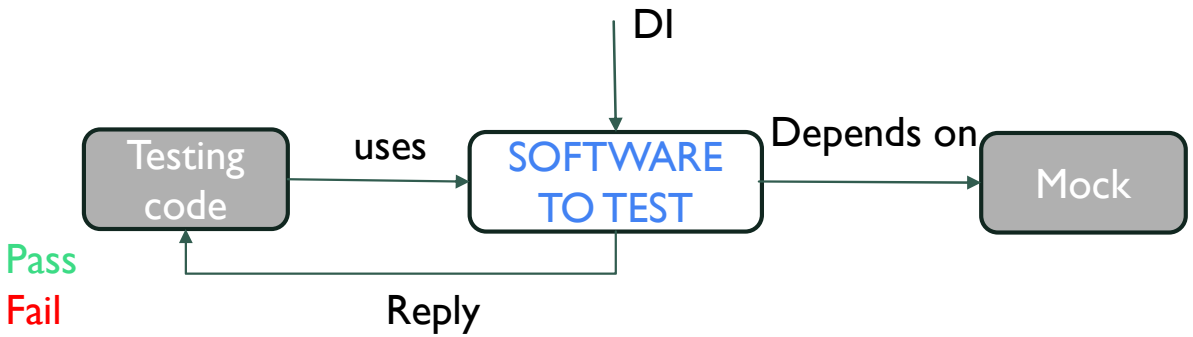
FUNCTIONAL TESTS



- Unit test tests a single component (like a UseCase), no device needed
- Integration test tests multiple components
- Instrumented tests (run on an emulator or a physical device)
- E2E (End-to-end) integration test tests a remote services
- A function test can either pass or fail

LOCAL UNIT TEST

- Local Unit Test : The goal is to check one chunk of code in isolation
- In Android Studio the folder test contains code that runs on the JVM (not the device or emulator)
- The @Test annotation allows to access to the classes in the source folder
- A test can either pass or fail

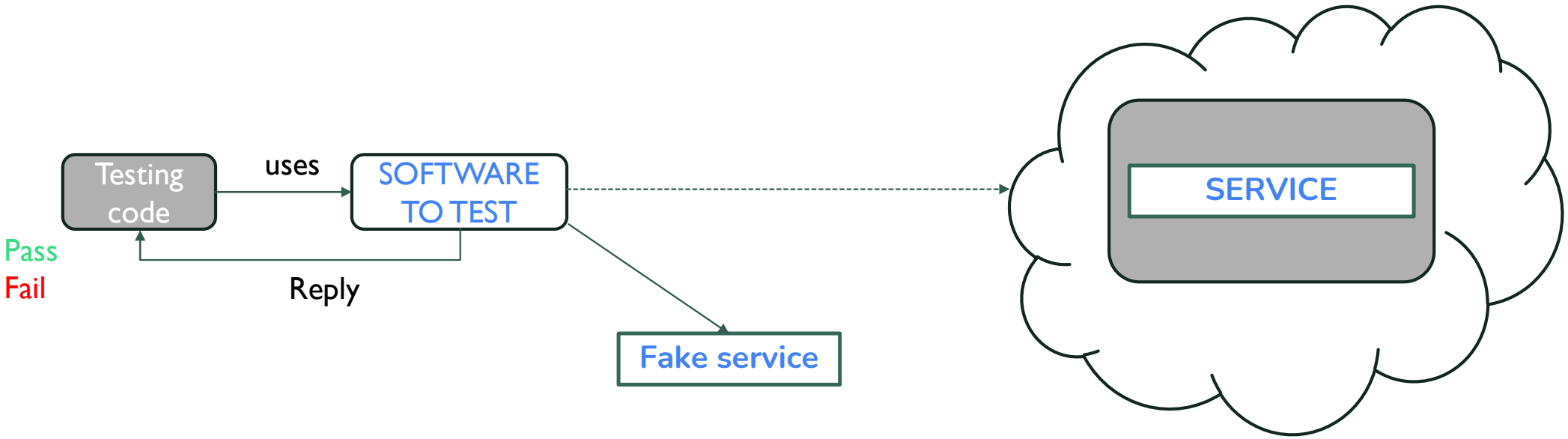


EXAMPLE OF UNIT TEST

```
class CheckWinnerUseCaseTest {
    private lateinit var checkWinnerUseCase: CheckWinnerUseCas
    @Before
    fun setUp() {
        checkWinnerUseCase = CheckWinnerUseCase()
    }
    @Test
    fun `invoke should return true for a winning row`() {
        val board = listOf(
            Player.X, Player.X, Player.X,
            Player.O, Player.NONE, Player.O,
            Player.NONE, Player.NONE, Player.NONE
        )
        val result = checkWinnerUseCase(board, Player.X)
        assertTrue(result)
    }
```

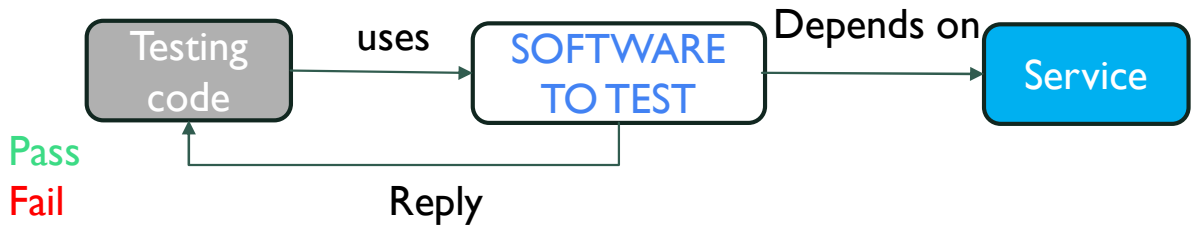
LOCAL UNIT TEST

- Useful to test that the remote service is correctly accessed before use it in the app
- The testing code verifies if an assertion is satisfied or not (pass or fail)
- A test can use fake data to test the code that parses the reply



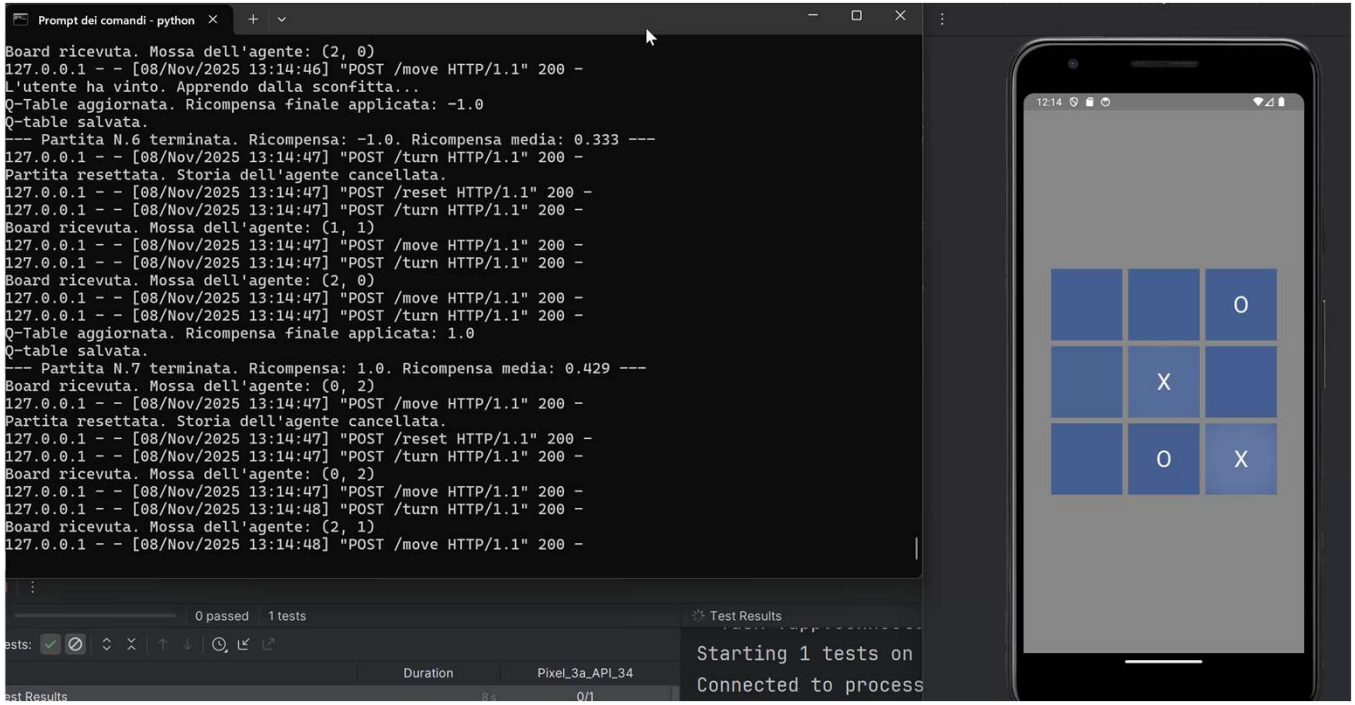
E2E TEST

- The remote service is running (perhaps after is local unit test !)
- When possible, it is useful to run the service on the local computer before moving it to a cloud
- Android emulator maps 10.0.2.2 to localhost 127.0.0.1 (it is an alias of the local host)
- For example, the address http://10.0.2.2:8080 points to a local web-server (or rest api) on port 8080



EXAMPLE OF INSTRUMENTED E2E TEST

- **Instrumented tests** run on an Android device, either physical or emulated.
- The app is built and installed alongside a *test app* that injects commands and reads the state.
- Instrumented tests are usually UI tests, launching an app and then interacting with it.



LAB-1: WRITE AN APP FOR THE TIC TAC TOE GAME

- Version 1: opponent is local and random
- Version 2: the opponent is remote and random
- Version 3: the opponent is remote and uses the RL

- Note: The agent can also generate python code
- Warning: free tier has per project limitation (generous)

TO DO

- Make move implemented in a docker container

LAB-2: REMOTE DATA FETCH EXAMPLE – EXCHANGE RATES API

- Scenario
- The app fetches real-time currency exchange rates from APIExchange sending a base currency (e.g., "EUR") as a request parameter.
- The server replies with a JSON object containing all rates.

TECHNICAL DESCRIPTION (CAN BE A PROMPT)

- UI (Jetpack Compose)
  - User selects the base currency
  - Triggers the action to fetch data (e.g., "Get Rates")
  - Observes the ViewModel state for updates
- ViewModel
  - Holds the UI state (Loading, Success, Error)
  - Calls the *UseCase* when user interaction occurs
  - Exposes immutable state to the UI (StateFlow / Compose state)
- Use Case (Domain Layer)
  - Encapsulates the business logic
  - data retrieval from the repository
  - May apply filtering, or transformations
- Repository (<<interface>>)
  - Delegates network requests to the API service
  - Converts API responses into domain model
- Remote Data Source proxy
  - Executes the HTTP request, e.g. Retrofit
    - Sends the base currency as a query parameter
  - Receives the JSON containing all exchange rates and convert to a data class