

FIRA Chovgan Simulation League

Student Programming Guide (U14 / U19)

Student Edition v1.3 | Chovgan SDK 1.0 — Final Student API & Executable Reference | 17 October 2026

FIRA Chovgan Simulation League Technical Committee

Welcome! Let's Program a Chovgan Robot

This guide is for students who want to write Python code and see what their simulated robot does. Start with Lesson 1. You do not need to understand every sensor on your first day.

In a match, Blue has B1, B2, B3 and B4. Yellow has Y1, Y2, Y3 and Y4. Each robot runs its own controller. All eight robots use the same basic programming interface. U14 teams play only other U14 teams, and U19 teams play only other U19 teams.

Before you start: The Chovgan SDK 1.0 student programming API is finalized. The executable SDK integration bundle can launch eight Python robot controllers and test these interfaces in its headless reference runtime. Running robot.py by itself in ordinary Python does not create a simulator Robot object. The official Chovgan build must connect the same API to its four-legged robot physics, sensors and Artificial Referee.

0. Explore the CHOVGAN Main Menu

Before writing code, get to know the simulator's main menu. The screen shown below has seven choices. Select a menu card to open that area. The descriptions are based on the visible labels and subtitles in the provided screenshot.

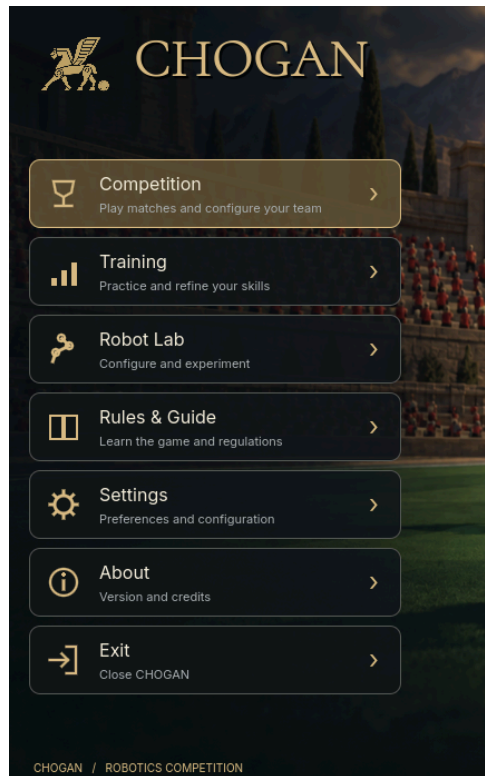


Figure 1 — Simulator main menu (image from an earlier build; the on-screen logo may use earlier branding).

Competition — Enter match setup, select U14 or U19, choose the teams, and reach the STRATEGY and READY stages. The pictured STRATEGY screen does not yet deploy student code.

Training — See six practice tiles, including Ball Control, Shooting, Passing, Movement, AI Testing, and Free Practice. Some dedicated drills are not yet available; check each tile's status.

Robot Lab — View a reference quadruped illustration and access the visible sensor-practice entry point. Motion Presets, Kick Test, Turning Test, and Gait Preview are marked unavailable in the supplied build.

Rules & Guide — Learn the game and regulations. Check this area to read in-app information about the game. The officially published competition rules remain authoritative.

Settings — Adjust graphics, camera, match presentation, audio, language, and controls as supported by the installed simulator.

About — Version and credits. Open this area to see information about CHOVGAN, including its version and credits.

Exit — Close CHOVGAN. Choose this when you want to leave the simulator.

Suggested learning order: Rules & Guide -> Robot Lab -> Training -> Competition. Settings and About can be opened whenever needed.

Remember: Menu labels explain where to begin, but they are not Python commands. The lessons below teach you how to write code for the robots.

0.1 Choose U14 or U19

From Competition, open Select Competition Division. The screen shows two cards: U14 (Beginner Competition) and U19 (Advanced Competition). Both describe four-robot teams. Only the U14 card says Coach Board available.



Figure 2 — Division selection (development screenshot; the application still displays its earlier on-screen branding).

Click Select U14 if your registered team competes in U14, or Select U19 if it competes in U19. These names describe the competition divisions, not two different programming APIs. The official rules determine eligibility; the screen does not replace registration.

0.2 Open Training to Practise

The Training screen shows six tiles: Ball Control, Shooting, Passing, Movement, AI Testing, and Free Practice. Read each tile and the information on the right before choosing a mode.

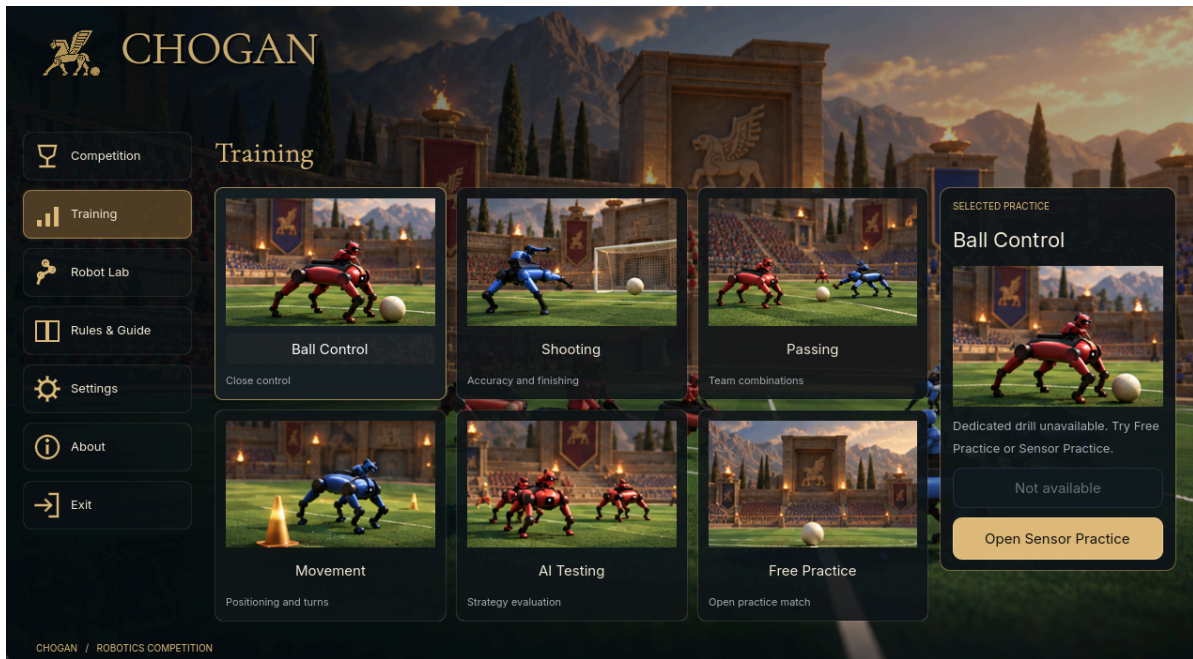


Figure 3 — Training screen (development build).

In the supplied screenshot, Ball Control is selected but says its dedicated drill is unavailable. It suggests Free Practice or Sensor Practice instead. The button Open Sensor Practice is visible. A tile being displayed does not prove that every practice mode is implemented in this build.

0.3 Explore Robot Lab

Robot Lab shows a quadruped robot reference illustration, a button labelled Open live sensor practice, and feature cards for Motion Presets, Kick Test, Turning Test, and Gait Preview.



Figure 4 — Robot Lab screen (development build; the displayed quadruped is labelled a reference illustration, not a live preview).

The four feature cards are each marked Not available in this build. Do not expect them to start a motion test here. The pictured robot is an illustration; it does not prove which actuators can be programmed through the SDK.

How do wheel-motor commands work on a four-legged robot? The public SDK uses the names 'left wheel motor' and 'right wheel motor'. These can be virtual movement controls: the simulator can translate the two requested speeds into walking forward, moving backward, or turning on a four-legged model. This requires a simulator-side motion adapter; there are no physical wheels in the illustrated model. The adapter, walking animation, and real movement must be tested before this behavior can be guaranteed.

0.4 Settings and the Match Display

Settings visibly includes Graphics (fullscreen, resolution, graphics preset, VSync), Camera, Match Presentation, Audio, Language, and Controls. These options change the user experience; the screenshot does not establish new commands for student robot code.

The match screen displays both teams' names, goals, a timer, and the current half (for example, H1). The supplied development screenshot shows a 2:59 clock in H1, which differs from the official competition rule of two two-minute halves. Follow the officially approved match clock, not old demonstration screenshots.

Some images were captured before the project was renamed to CHOVGAN. The text of this guide uses the corrected name. Menu placement and feature availability may change in future builds.

1. Your First Goal: Make the Robot Move

Create a file named robot.py in your team project. Copy this entire example. It sends forward-motion commands for about two seconds and then sends stop commands using the SDK's two virtual drive interfaces.

```
from controller import Robot

TIME_STEP = 32
robot = Robot()

left = robot.getDevice("left wheel motor")
right = robot.getDevice("right wheel motor")

left.setPosition(float("inf"))
right.setPosition(float("inf"))

start = robot.getTime()

while robot.step(TIME_STEP) != -1:
    if robot.getTime() - start < 2.0:
        left.setVelocity(3.0)
        right.setVelocity(3.0)
    else:
        left.setVelocity(0.0)
        right.setVelocity(0.0)
```

What should happen?

The controller requests speed 3 on both named motor interfaces for two seconds of simulation time, then requests speed 0. If the quadruped motion adapter is implemented as specified, the robot walks forward and stops. These numbers are virtual drive commands, not physical wheel speeds or metres per second. Confirm this behavior in the released simulator.

What do these lines mean?

- `from controller import Robot` — connects your program to the simulator-provided robot interface.
- `robot = Robot()` — gives your program access to its assigned robot.
- `robot.getDevice("left wheel motor")` — finds the SDK's named left-drive interface. On a quadruped, this may be a virtual control rather than a physical wheel.
- `setPosition(float("inf"))` — puts the motor in velocity-control mode.
- `setVelocity(3.0)` — requests a virtual drive speed of 3; it does not mean 3 metres per second or direct control of one leg.
- `robot.step(32)` — waits for the next simulation update; 32 is the documented step interval in milliseconds.
- `robot.getTime()` — returns the time in the simulation, measured in seconds.
- `setVelocity(0.0)` — tells the motor to stop.

Try it: Change both 3.0 values to 5.0. Does the robot move faster? Then change the duration 2.0 to 4.0. Do not use values above the documented motor range of -10 to +10.

2. Speed, Backward Motion, and Turning

Once you have set up the motors as in Lesson 1, you can change their speeds. These are commands, not complete programs by themselves.

```
left.setVelocity(0.0)
right.setVelocity(0.0)    # Stop

left.setVelocity(3.0)
right.setVelocity(3.0)    # Move forward (documented convention)

left.setVelocity(-3.0)
right.setVelocity(-3.0)   # Move backward (documented convention)

left.setVelocity(3.0)
right.setVelocity(-3.0)   # Turn in place

left.setVelocity(2.0)
right.setVelocity(4.0)    # Follow a curved path
```

Why? Your code sends two movement-speed commands through the SDK. On a four-legged Chovgan robot, these names may represent virtual left/right drive controls, rather than actual wheels. The simulator must translate them into a suitable gait and turn rate. The exact turning direction and speed must be verified in the official build.

Important: Calling `setVelocity` once requests a speed; it does not move the simulator forward by itself. Your controller still needs a working `robot.step(...)` loop. The SDK contract defines a virtual motor-command range of -10 to +10; these values are not physical wheel speeds. The final simulator maps them to quadruped motion.

3. Robot Names: B1, B2, B3, B4 and Y1, Y2, Y3, Y4

Each controller should ask the simulator which robot it belongs to. Never hardcode 'B1' into code that is intended to work on every player.

```
from controller import Robot

robot = Robot()
name = robot.getName()

team = name[0]          # "B" or "Y"
player_number = int(name[1:]) # 1, 2, 3, or 4

print("My robot is:", name)
```

What should happen? B3 prints 'B3' and has player_number equal to 3. A Yellow robot might print 'Y2'. The simulator assigns the name; students do not choose it from within their controller.

4. Sensors: How Does the Robot Notice an Obstacle?

The documented SDK has four distance sensors: front, back, left and right. Their values are on a compatibility scale: a smaller value usually means a closer obstacle. The number is not automatically a distance in centimetres.

Example: Move until something is close in front

```
from controller import Robot

robot = Robot()
TIME_STEP = 32

left = robot.getDevice("left wheel motor")
right = robot.getDevice("right wheel motor")
front = robot.getDevice("distancesensor front")

left.setPosition(float("inf"))
right.setPosition(float("inf"))
front.enable(TIME_STEP)

while robot.step(TIME_STEP) != -1:
    reading = front.getValue()

    if reading < 300:
        left.setVelocity(0.0)
        right.setVelocity(0.0)
    else:
        left.setVelocity(3.0)
        right.setVelocity(3.0)
```

What should happen?

When the front reading falls below the example threshold 300, the robot stops. Otherwise, it requests forward motion. The threshold 300 is a raw sensor value chosen for learning; it does NOT mean 300 centimetres or a guaranteed safe distance.

Try it: Replace 300 with 500 and compare how early the robot stops. Ask a teacher or coach before testing any physical robot; this guide is written for the simulation.

5. GPS: Where Am I on the Field?

GPS lets the controller read its location in the simulator. The SDK describes three values: x, y and height. x and y are field coordinates, not GPS latitude and longitude.

```
from controller import Robot
```

```
robot = Robot()
gps = robot.getDevice("gps")
gps.enable(32)
```

```
while robot.step(32) != -1:
    x, y, height = gps.getValues()
    # The robot now knows its own position.
    # You can use x and y in your strategy.
```

What happens? Each update gives the robot its position. For example, your code can compare x with a target x coordinate. The final field dimensions, units and coordinate origin must be confirmed by the Technical Committee.

6. Compass: Which Way Am I Facing?

A compass provides a direction vector. Do not assume its first number is already an angle in degrees. For ordinary beginners, first read the values. Calculating a heading is an optional advanced step.

```
from controller import Robot
```

```
robot = Robot()
compass = robot.getDevice("compass")
compass.enable(32)
```

```
while robot.step(32) != -1:
    direction = compass.getValues()
    # direction has three components.
    # These values describe orientation.
```

Optional: calculate an angle

```
import math
```

```
v = compass.getValues()
heading = math.atan2(v[0], v[1]) + math.pi / 2
```

```
if heading < -math.pi:
    heading += 2 * math.pi
```

This matches the current helper calculation and gives an angle in radians. It is not a complete guarantee that the angle is normalized to every value in $[-\pi, \pi]$. The Chovgan field orientation must be calibrated before you convert angles into movement decisions.

7. Finding the Ball

The ball receiver gives your robot observations of the ball. A message can contain the ball's local direction and a signal-strength value. A stronger signal commonly means the ball is nearer; it is not a direct distance in metres.

```
from controller import Robot

robot = Robot()
ball = robot.getDevice("ball receiver")
ball.enable(32)

while robot.step(32) != -1:
    if ball.getQueueLength() > 0:
        direction = ball.getEmitterDirection()
        strength = ball.getSignalStrength()

        print("Ball direction:", direction)
        print("Signal strength:", strength)

    ball.nextPacket()
```

What should happen?

When a ball observation is waiting, the program reads its direction and strength. It then calls `nextPacket()` to remove that observation from the queue. If no observation is queued, the code waits for the next simulation step.

Do not assume `direction[0]` or `direction[1]` always means 'left' without checking the simulator's coordinate conventions. This receiver does not automatically report the ball's global x and y. A ball-control or striking actuator is not specified by the reviewed SDK.

8. Communication: Let Your Robots Share Information

Robots on the same team can send information to one another. They must not send messages to the other team. One message can be text, or text containing a JSON dictionary.

Example: Send your name to your teammates

```
import json
from controller import Robot

robot = Robot()
sender = robot.getDevice("team emitter")
receiver = robot.getDevice("team receiver")
receiver.enable(32)

last_send = -1.0
```

```

while robot.step(32) != -1:
    now = robot.getTime()

    if now - last_send >= 1.0:
        message = {
            "robot": robot.getName(),
            "status": "ready"
        }
        sender.send(json.dumps(message))
        last_send = now

    while receiver.getQueueLength() > 0:
        text = receiver.getString()
        receiver.nextPacket()
        data = json.loads(text)
        print("Teammate says:", data)

```

What should happen?

Approximately once per simulation second, the robot sends a small JSON message to its team. When teammate messages arrive, the robot reads and removes them from its receive queue. The documented maximum payload is 4,096 bytes. Do not rely on messages from different teammates arriving in a fixed order.

9. Referee Messages: Learn What Happened in the Match

The Artificial Referee handles goals, own goals, fouls, aerial ball restarts and the match clock. A read-only 'supervisor receiver' lets your program read referee messages. The final fields in those messages have not yet been approved.

```

from controller import Robot

robot = Robot()
referee = robot.getDevice("supervisor receiver")
referee.enable(32)

while robot.step(32) != -1:
    while referee.getQueueLength() > 0:
        message = referee.getString()
        print("Referee message:", message)
        referee.nextPacket()

```

What should happen? If the runtime sends a referee message, the program prints it. Do not write strategies that depend on a specific key such as 'score' or 'foul' until the official referee message format is published.

10. Use ChovganRobot to Write Less Code

The finalized SDK 1.0 starter includes chovgan_robot.py and its ChovganRobot helper. A helper is a Python class that already knows how to find common sensors and read their data. It does not invent new robot powers.

Useful helper methods

- `get_gps_coordinates()` — reads the robot's x and y coordinates.
- `get_compass_heading()` — calculates the heading in radians.
- `get_sonar_values()` — returns a dictionary with front/back/left/right readings.
- `is_new_ball_data()` — checks whether a ball observation is queued.
- `get_new_ball_data()` — reads and consumes a ball observation.
- `is_new_team_data()` — checks for a teammate message.
- `get_new_team_data()` — reads and consumes a teammate message.
- `send_data_to_team(value)` — broadcasts your value to teammates by placing it in a JSON message under the key `robot_id`.
- `is_new_data()` and `get_new_data()` — check for and read referee-related packets.

Example of using the helper

```
from chovgan_robot import ChovganRobot, TIME_STEP

class MyChovganRobot(ChovganRobot):
    def run(self):
        while self.robot.step(TIME_STEP) != -1:
            if self.is_new_ball_data():
                ball = self.get_new_ball_data()
                print("Ball strength:", ball["strength"])

            if self.is_new_team_data():
                message = self.get_new_team_data()
                print("Team data:", message)
```

This is a player class, not a complete `robot.py` launcher. The SDK sample project must create a `Robot` object, pass it to your player class, and call `run()`. Do not call `get_new_ball_data()` when no packet is queued.

A naming detail: in the current helper, `send_data_to_team(robot_id)` is not a function for choosing one recipient. Its argument is the content to broadcast, which is wrapped in a JSON field named `robot_id`. For beginners, the direct team emitter example in Lesson 8 may be easier to understand.

11. What Does a Four-Robot Team Look Like?

A Chovgan match contains four robots per team, but each robot has its own active Python controller. A project may use one launcher and four strategy classes. Students do not need four different sensor APIs: all players use the same devices.

```
chovgan_team/
robot.py          # Starts the correct player controller
chovgan_robot.py  # Shared SDK helper
robot1.py         # Code for player 1
robot2.py         # Code for player 2
robot3.py         # Code for player 3
robot4.py         # Code for player 4
utils.py          # Optional shared calculations
```

How the launcher chooses a player

```
from controller import Robot
from robot1 import ChovganPlayer1
```

```

from robot2 import ChovganPlayer2
from robot3 import ChovganPlayer3
from robot4 import ChovganPlayer4

```

```

PLAYERS = {
    1: ChovganPlayer1,
    2: ChovganPlayer2,
    3: ChovganPlayer3,
    4: ChovganPlayer4
}

```

```

robot = Robot()
number = int(robot.getName()[1:])
player = PLAYERS[number](robot)
player.run()

```

This launcher is a documented example, not a guarantee of the final submission folder format. The four ChovganPlayer classes must exist in your player files. The official runtime must confirm how it starts these controllers.

11.1 Submit One ZIP File for Your Whole Team

For the proposed official Chovgan submission format, prepare ONE ZIP file containing code for all four players. The final competition build must support loading the submitted code, and the Technical Committee will confirm the official upload channel, accepted ZIP structure, and folder name.

What should be inside the ZIP?

```

ChovganTeam.zip
chovgan_team/
  robot.py           # Starts the correct player
  chovgan_robot.py   # Shared ChovganRobot helper
  robot1.py          # Player 1
  robot2.py          # Player 2
  robot3.py          # Player 3
  robot4.py          # Player 4
  utils.py           # Optional shared functions
  team_name.txt      # Your team's name (required in this proposal)
  logo.png           # Optional team logo

```

When you open the ZIP, chovgan_team/ and team_name.txt must appear immediately at the top level. Do not put them inside another ChovganTeam/ or controllers/ folder. The folder name chovgan_team/ is proposed and must be supported by the official Chovgan launcher before the format becomes mandatory.

Follow these six steps

1. Create your chovgan_team folder and put robot.py, chovgan_robot.py, and robot1.py through robot4.py inside it.
2. Add utils.py or your own extra Python modules if your program imports them.
3. Create a plain-text file called team_name.txt next to the chovgan_team folder. Write your team name on one line.

4. Optionally add logo.png next to team_name.txt. The final simulator must confirm whether team logos are displayed.
5. Select chovgan_team/, team_name.txt, and (if used) logo.png, and compress them together into one .zip archive.
6. Open the ZIP to check its contents. Submit it using the officially approved code-upload procedure and check that the simulator confirms validation and starts the submitted controllers.

What should happen after an official team-code loader is released?

A proposed Chovgan launcher uses robot.py to read a robot name (B1–B4 or Y1–Y4), select the corresponding player class, and call run(). The official competition build must confirm that the code loader invokes this launcher and runs all four submitted robot controllers, rather than a built-in demonstration strategy.

Check before uploading: (1) all four player files are present; (2) robot.py imports them correctly; (3) chovgan_robot.py is included when your code uses it; (4) team_name.txt is a real text file, not team_name.txt.txt; (5) the ZIP opens without errors; (6) there is no extra parent directory or controllers/ folder. The simulator supplies the controller Python package; do not include your own replacement.

Competition release note: The earlier reference submission policy used a 10 MB ZIP limit and allowed team names up to 32 characters. For Chovgan, these are PROPOSED limits, not approved requirements. The Technical Committee must decide ZIP size, team-name length, the actual folder name, logo support, upload location, deadline, and validation policy before the guide becomes binding.

11.2 Quick Submission Check: Keep It Simple

Each robot uses the same submitted team folder. You do not need a separate ZIP for B1, B2, B3 and B4. The simulator assigns the robot identity. Your code selects the appropriate strategy.

A tiny example of team_name.txt

Chovgan Explorers

The file is plain text. Write only your team's name. The example is not a registration instruction, and the Technical Committee will confirm any naming rules.

What belongs in your ZIP?

- Include the Python files your player code actually imports. utils.py is optional, but if robot2.py imports it, it must be present.
- Keep the team helper chovgan_robot.py with the player scripts when you use ChovganRobot. Do not include a second copy of the simulator's controller/ package.
- Do not import your player code from robot.py into robot1.py or utils.py; this can create a circular import. Put shared helper functions in utils.py instead.
- Keep the code that is actually needed for your team. You normally do not need the entire simulator, its referee program, or its field files in your team ZIP.

- Prefer Python's standard library. Ask the organizers before relying on extra modules, because the exact versions available in Chovgan have not been confirmed.

If the organizer rejects your ZIP

First, open the ZIP and inspect its top level. The proposed format contains `chovgan_team/` and `team_name.txt` directly, plus optional `logo.png`. Check the spelling of `robot.py`, `robot1.py`, `robot2.py`, `robot3.py` and `robot4.py`. Then verify that each player module defines the class imported by `robot.py`. If the official event specifies a different folder or launcher name, follow that official instruction instead.

Fair play and student-written code

Write and test your own strategy. You may learn from examples and openly licensed projects, but do not submit someone else's work as your own. Follow the license terms of any reused source code. The reference event used code-similarity screening and later publication of submitted code; whether Chovgan will require those practices must be decided by its organizers and communicated to teams in advance.

Organizer approval still needed: Finalize the submission-folder name, exact launcher filename, upload platform, ZIP size limit, naming limits, optional logo behavior, supported Python packages, code originality policy, and any later publication of submitted source code. Until then, this section is a student-friendly proposed format, not a final event regulation.

11.3 Strategy Deployment — Choosing and Checking Your Code

Competition setup shows four stages: MATCH, TEAMS, STRATEGY (03), and READY (04). The screenshot below shows an earlier development build of the STRATEGY stage. The final build must verify and execute the code selected for the match.

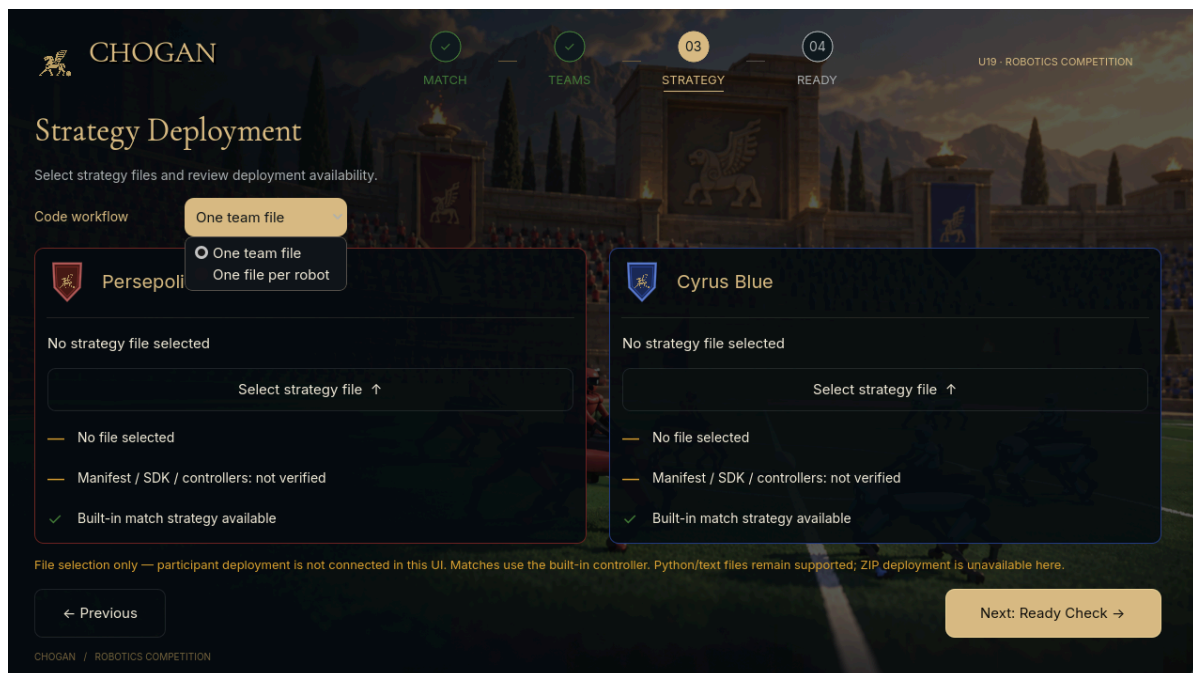


Figure 5 — Strategy Deployment screen in the supplied development build. The app in this screenshot still shows its previous on-screen branding.

How to use the visible screen

1. Open Competition from the main menu and choose your division: U14 or U19. Complete the earlier match and teams stages until you reach STRATEGY (03).
2. Open Code workflow. Two choices are visible: One team file and One file per robot. They are interface options in this build; neither proves that code will be executed.
3. In each team's panel, use Select strategy file to choose a file. The screenshot shows separate panels for Persepolis Red and Cyrus Blue.
4. Read the status messages beneath the selector. No file selected and Manifest / SDK / controllers: not verified mean the selected strategy has not been verified as deployable.
5. Built-in match strategy available means the simulator has an internal match strategy. It does not mean your own Python code has been loaded.
6. Next: Ready Check moves to the next setup stage. Do not interpret reaching READY as proof that a participant controller has been installed or executed.

Important: file selection is NOT code execution

The screenshot was captured before student-code deployment was finished. Its yellow warning reflects that earlier development stage, not the required behavior for the competition release. In the official build, selecting a file must be followed by successful validation and actual execution of the submitted controllers.

Prepare your Python source files and team ZIP using the organizer's approved format. Before a match, check the simulator's validation result and confirm that each robot is running the intended participant controller. If the file is rejected, fix the reported error before continuing.

Two formats that must not be confused

One team file and One file per robot are workflow choices visible in the STRATEGY screen. The proposed team ZIP containing robot.py and robot1.py–robot4.py is a separate packaging convention described in Section 11.1. The official build must make clear which files each option accepts and how they are mapped to all four robots.

For teachers: verify supported filename extensions, ZIP extraction, module imports, controller startup for all four players, match logs, and clear success/failure feedback in a live build before instructing students to submit code here.

12. U14 and U19: What Is Different?

The two divisions compete separately. Both use the same four-robot controller interface and the same baseline game rules. The official Coach Board exists only in U14. U19 does not have this feature.

Coach Board behavior has not been finalized. This guide intentionally gives no Coach Board commands, menus, or permissions yet. Students should not guess an API or attempt to implement an unofficial substitute.

13. Five Small Challenges

- Challenge 1 (start here): Change the first program so the robot moves for four seconds, then stops.
- Challenge 2: Make the robot reverse for one second by using two negative wheel speeds.

- Challenge 3: Read the front distance sensor. Make the robot stop at a chosen raw threshold. Test two threshold values.
- Challenge 4: Print your robot's name and its GPS coordinates. Which values change when it moves?
- Challenge 5 (advanced): Send a JSON message with your robot name and the latest ball-signal strength. Decide how to handle the case when no ball observation arrives.

Suggested path: U14 beginners can start with Challenges 1–4. U19 students and experienced U14 students can also work on Challenge 5 and combine sensors with team strategy. These are learning suggestions, not different competition rules.

14. Common Problems and How to Fix Them

- Error importing 'controller': Your code is probably running outside the simulator. Use the official Chovgan launcher.
- The robot does not move: Check both motor names, velocity mode, nonzero speed, the step loop, and whether play is active.
- The robot moves in the wrong direction: Confirm motor orientation and the field axes in the released simulator.
- A sensor does not update: Check that enable(32) was called and a simulator step has occurred.
- The ball message repeats: Consume the current packet using nextPacket().
- Teammate messages are unreadable: Agree on one message format and check your JSON decoding.
- Player 4 does not start: Check robot4.py and the player-number mapping.
- The Coach Board is missing: It is U14-only; its technical contract is still awaiting publication.

15. Mini Dictionary

- Controller: The Python program that decides what one robot should do.
- SDK: The set of programming interfaces and helpers you use to control robots.
- Device: A named motor, sensor, emitter or receiver provided by the simulator.
- Simulation step: One update in simulated time. Use robot.step(TIME_STEP) to synchronize.
- Sensor: A device that reports measurements, such as position or distance.
- Packet / queue: A message and a list of messages waiting to be read.
- Robot identity: One of B1–B4 or Y1–Y4.
- Artificial Referee: The official system that decides match events automatically.

Appendix A. Exact Names You Must Spell Correctly

"gps"	"compass"
"ball receiver"	"supervisor receiver"
"team emitter"	"team receiver"
"distancesensor front"	"distancesensor back"
"distancesensor left"	"distancesensor right"
"left wheel motor"	"right wheel motor"

<code>robot.getName()</code>	Your robot's ID
<code>robot.getTime()</code>	Simulation time (seconds)
<code>robot.getDevice(name)</code>	Find a device
<code>robot.step(32)</code>	Wait for the next update
<code>sensor.enable(32)</code>	Start sensor updates
<code>gps.getValues()</code>	[x, y, height]
<code>compass.getValues()</code>	Direction vector
<code>distance_sensor.getValue()</code>	Raw distance reading
<code>receiver.getQueueLength()</code>	Number of waiting packets
<code>receiver.getString()</code>	Read current packet as text
<code>receiver.nextPacket()</code>	Move to the next packet
<code>ball.getEmitterDirection()</code>	Local ball direction
<code>ball.getSignalStrength()</code>	Ball signal strength
<code>emitter.send(text)</code>	Send a teammate message
<code>motor.setPosition(float("inf"))</code>	Velocity-control mode
<code>motor.setVelocity(speed)</code>	Request a wheel speed

Appendix B. Important Technical Limits

The candidate SDK documents `TIME_STEP = 32` milliseconds, nominal motor speeds from -10 to +10, and a team-message limit of 4,096 bytes. It also describes the ball receiver as a FIFO queue and distance sensor readings on a compatibility scale. These are documented candidate values, not independently certified measurements from the final Chovgan match runtime.

Never use private simulator controls, unsupported devices, direct ball-state modifications, or unauthorized external communication. The Artificial Referee decides legal goals, fouls, restarts, and the match result. Students may interact with the ball through officially permitted robot actions, but no separate Chovgan strike or possession method has yet been published in the reviewed SDK.

Appendix C. What Still Needs Official Confirmation?

- The exact simulator version, student installation procedure, Python version and official team upload format.
- A successful full-match test with all eight robot controllers.
- Final motor behavior, permitted ball-contact controls, field dimensions and sensor calibration.
- The message fields sent by the Artificial Referee.
- The functions and permissions of the U14 Coach Board.
- Runtime resource limits, official error handling, and release approval.

Teachers and Technical Committee: Chovgan SDK 1.0 includes a working eight-controller Python reference runtime. Its acceptance suite passes eight SDK contract tests and six integration tests, including real Python processes, ZIP validation and team communication. The published student API is finalized. Production integration must connect these same interfaces to the official simulator's quadruped, sensors, ball interactions, clock and Artificial Referee, and should retain a real-match acceptance log.