KHEL AI MVP

AI Agent Build Guide for Django Webapp

Step-by-step implementation guide with repo structure, code changes, settings, tools, memory, prompts, API integration, live banners, testing, and deployment checklist

GitHub Repository: HimanshuKhale/livestream_ksjd

Clone URL

Prepared for Khel AI MVP / Cricket Analytics and Predictions workflow

How to use this document

Use this PDF as the implementation checklist for adding a robust AI Agent to khel_ai_mvp. Start with requirements and settings, then add database memory, tool registry, validators, frontend chat, live-banner integration, and finally Sprint 3 model tools.

Live Link Index

Resource	Clickable link	URL
Main GitHub repo	Open link	https://github.com/HimanshuKhale/livestream_ksjd
Clone URL	Open link	https://github.com/HimanshuKhale/livestream_ksjd.git
Django documentation	Open link	https://docs.djangoproject.com/
OpenAI API documentation	Open link	https://platform.openai.com/docs
OpenAI Responses API guide	Open link	https://platform.openai.com/docs/guides/responses
Requests documentation	Open link	https://requests.readthedocs.io/
python-dotenv PyPI	Open link	https://pypi.org/project/python-dotenv/
Render documentation	Open link	https://docs.render.com/
django-redis documentation	Open link	https://github.com/jazzband/django-redis
Bowler Momentum API	Open link	https://bowler-momentum-api-3.onrender.com/api/bowler-momentum/
Student 1 Sprint 1 API base	Open link	https://khel-ai-st1-sp1-1.onrender.com
Student 2 Sprint 1 API base	Open link	https://khel-ai-st2-sp1.onrender.com
Student 3 Sprint 1 API base	Open link	https://khel-ai-st3-sp1.onrender.com
Student 4 Sprint 1 API base	Open link	https://khel-ai-st4-sp1.onrender.com
Student 5 Sprint 1 API base	Open link	https://khel-ai-st5-sp1.onrender.com
Student 1 Sprint 2 API base	Open link	https://khel-ai-st1-sp2.onrender.com

Student 2 Sprint 2 API base	Open link	https://khel-ai-st2-sp2.onrender.com
Student 3 Sprint 2 API base	Open link	https://khel-ai-st3-sp2.onrender.com

Table of Contents

1. Agent Architecture Overview

In Khel AI MVP, the AI Agent is not merely a chatbot. A chatbot replies with text. An agent reads the current match state, decides whether a cricket analytics tool is required, calls the appropriate local or external API, interprets the result, optionally creates a live broadcast banner, and remembers the interaction for future context.

```
Admin/User Message
        ↓
Django View: khel_ai_agent_api
        ↓
Agent Runner: run_khel_ai_agent()
        ↓
Context Builder: build_match_context()
        ↓
System Prompt + JSON Instruction + Match Context + Recent Memory
        ↓
LLM returns structured JSON decision
        ↓
Django validates and routes decision
        ↓
Tool Registry selects Python tool
        ↓
Tool calls local services or external Render APIs
        ↓
Optional live banner is written to Django cache
```

↓

Memory layer stores user, assistant, and tool metadata

↓

Frontend receives JSON response

↓

Admin chat displays answer / live page displays banner

Recommended MVP pattern

Use a JSON-router agent first. The model returns a strict JSON decision, and Django chooses the actual Python tool. This is easier to debug than native function calling and safer for an internship or student-built API ecosystem.

2. Current Repo State and Existing Files

The current repository is a Django MVP named khel_ai_mvp with a matches app. The existing app already includes live scoring, match and innings models, ball events, external analytics clients, live banner cache usage, and an initial ai_agent package.

Repository: https://github.com/HimanshuKhale/livestream_ksjd

Current high-level repo structure

```
livestream_ksjd/
├── manage.py
├── requirements.txt
├── db.sqlite3
├── .env.example
├── AGENTS.md
├── README.md
├── khel_ai_mvp/
│   ├── settings.py
│   ├── urls.py
│   ├── asgi.py
│   └── wsgi.py
└── matches/
    ├── models.py
    ├── views.py
    ├── urls.py
    ├── forms.py
    ├── services.py
    ├── scoring.py
    ├── api_clients.py
    ├── analytics_payloads.py
    ├── sprint2_payloads.py
    ├── student2_sprint2_payloads.py
    ├── student3_sprint2_payloads.py
    └── ai_agent/
        ├── agent.py
        ├── context_builder.py
        ├── memory.py
        ├── prompts.py
```

```
├── schemas.py
└── tools.py
```

Important existing patterns already present

- settings.py already loads dotenv and reads OPENAI_API_KEY and KHEL_AI_AGENT_MODEL from environment variables.
- settings.py already stores base URLs for Student 1, 2, 3, 4, and 5 APIs and Sprint 2 APIs.
- khel_ai_mvp/urls.py includes matches.urls at the root, so routes declared inside matches/urls.py become live.
- matches/urls.py already contains an agent API route: api/agent/<int:match_id>/.
- matches/views.py already imports run_khel_ai_agent and exposes khel_ai_agent_api.
- matches/ai_agent/tools.py already contains tool functions for scorecard, Student 1 batting, Student 2 bowling, Student 3 all-rounder metrics, and temporary live banners.
- matches/ai_agent/memory.py is currently a stub and should be upgraded to database-backed memory.

3. Required Dependencies and Environment Settings

The existing requirements file is too small for the imports used by the agent. The project imports dotenv, requests, and OpenAI SDK, so those packages must be present.

requirements.txt

```
asgiref==3.11.1
Django==6.0.3
sqlparse==0.5.5
tzdata==2025.3
python-dotenv>=1.0.1
openai>=2.0.0
requests>=2.32.0
```

Install dependencies:

```
pip install -r requirements.txt
```

.env file

```
SECRET_KEY=replace-this-in-production
DEBUG=True
ALLOWED_HOSTS=127.0.0.1,localhost

OPENAI_API_KEY=sk-your-real-key-here
KHEL_AI_AGENT_MODEL=gpt-5.4-nano

STUDENT1_SPRINT2_API_BASE_URL=https://khel-ai-st1-sp2.onrender.com
STUDENT2_SPRINT2_API_BASE_URL=https://khel-ai-st2-sp2.onrender.com
STUDENT3_SPRINT2_API_BASE_URL=https://khel-ai-st3-sp2.onrender.com

EXTERNAL_ANALYTICS_API_TIMEOUT=90
LIVE_ANALYTICS_API_TIMEOUT=90
```

```
LIVE_INFOGRAPHIC_TTL_SECONDS=20
```

Security note

Never commit the real .env file. Keep API keys in Render environment variables, local .env files, or a secret manager. Only .env.example should go to GitHub.

settings.py improvements

```
from pathlib import Path
import os
from dotenv import load_dotenv

load_dotenv()
BASE_DIR = Path(__file__).resolve().parent.parent

SECRET_KEY = os.environ.get(
    "SECRET_KEY",
    "django-insecure-dev-only-change-me"
)

DEBUG = os.environ.get("DEBUG", "True").lower() == "true"

ALLOWED_HOSTS = os.environ.get(
    "ALLOWED_HOSTS",
    "127.0.0.1,localhost"
).split(",")

OPENAI_API_KEY = os.environ.get("OPENAI_API_KEY", "")
KHEL_AI_AGENT_MODEL = os.environ.get("KHEL_AI_AGENT_MODEL", "gpt-5.4-nano")

STUDENT1_SPRINT2_API_BASE_URL = os.environ.get(
    "STUDENT1_SPRINT2_API_BASE_URL",
    "https://khel-ai-st1-sp2.onrender.com"
)
STUDENT2_SPRINT2_API_BASE_URL = os.environ.get(
    "STUDENT2_SPRINT2_API_BASE_URL",
    "https://khel-ai-st2-sp2.onrender.com"
)
STUDENT3_SPRINT2_API_BASE_URL = os.environ.get(
    "STUDENT3_SPRINT2_API_BASE_URL",
    "https://khel-ai-st3-sp2.onrender.com"
)

EXTERNAL_ANALYTICS_API_TIMEOUT = int(os.environ.get("EXTERNAL_ANALYTICS_API_TIMEOUT", "90"))
LIVE_ANALYTICS_API_TIMEOUT = int(os.environ.get("LIVE_ANALYTICS_API_TIMEOUT", "90"))
LIVE_INFOGRAPHIC_TTL_SECONDS = int(os.environ.get("LIVE_INFOGRAPHIC_TTL_SECONDS", "20"))
```

4. Target Repository Structure

Keep the AI agent as a Python package inside the matches app. This keeps the MVP simple while still separating prompts, memory, tools, schemas, validators, and orchestration logic.

```
matches/
└── ai_agent/
    ├── __init__.py
    ├── agent.py            # Main orchestration logic
    ├── context_builder.py  # Converts Django match data into agent-readable JSON
    ├── memory.py           # Saves and retrieves conversation history
    ├── prompts.py          # System prompts and JSON instructions
    ├── registry.py         # Maps metric names to tool functions
    ├── schemas.py          # Dataclasses for responses
    ├── tools.py            # Local and external API tool functions
    ├── validators.py       # Cleans JSON output and user/tool inputs
    └── tests/
        ├── test_agent.py
        ├── test_tools.py
        └── test_context_builder.py
```

5. Database Memory Models

The current memory.py does not store anything in the database. Add two models: AgentConversation and AgentMessage. This lets the agent remember prior messages, previous tool outputs, and banner actions for each match or innings.

```
# matches/models.py

class AgentConversation(models.Model):
    match = models.ForeignKey(
        Match,
        on_delete=models.CASCADE,
        related_name="agent_conversations"
    )
    innings = models.ForeignKey(
        Innings,
        on_delete=models.SET_NULL,
        null=True,
        blank=True,
        related_name="agent_conversations"
    )
    title = models.CharField(max_length=200, blank=True)
    created_by = models.ForeignKey(
        "auth.User",
        on_delete=models.SET_NULL,
        null=True,
        blank=True,
        related_name="khel_agent_conversations"
    )
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
```

```
    is_active = models.BooleanField(default=True)

    class Meta:
        ordering = ["-updated_at"]

    def __str__(self):
        return self.title or f"Agent Conversation for {self.match}"


class AgentMessage(models.Model):
    ROLE_CHOICES = [
        ("user", "User/Admin"),
        ("assistant", "Assistant"),
        ("tool", "Tool"),
        ("system", "System"),
    ]

    conversation = models.ForeignKey(
        AgentConversation,
        on_delete=models.CASCADE,
        related_name="messages"
    )
    role = models.CharField(max_length=20, choices=ROLE_CHOICES)
    content = models.TextField()
    metadata = models.JSONField(default=dict, blank=True)
    created_at = models.DateTimeField(auto_now_add=True)

    class Meta:
        ordering = ["created_at"]

    def __str__(self):
        return f"{self.role}: {self.content[:60]}"
```

Run migrations:

```
python manage.py makemigrations matches
python manage.py migrate
```

Admin registration

```
# matches/admin.py
from django.contrib import admin
from .models import AgentConversation, AgentMessage

@admin.register(AgentConversation)
class AgentConversationAdmin(admin.ModelAdmin):
    list_display = ("id", "match", "innings", "title", "created_by", "is_active", "updated_at")
    list_filter = ("is_active", "created_at")
    search_fields = ("title", "match__title")


@admin.register(AgentMessage)
class AgentMessageAdmin(admin.ModelAdmin):
    list_display = ("id", "conversation", "role", "created_at")
```

```
    list_filter = ("role", "created_at")
    search_fields = ("content",)
```

Database-backed memory.py

```
# matches/ai_agent/memory.py
from typing import Optional, Dict, Any, List

from matches.models import Match, Innings, AgentConversation, AgentMessage


def get_or_create_conversation(
    match: Match,
    innings: Optional[Innings] = None,
    user=None,
    conversation_id: Optional[int] = None,
) -> AgentConversation:
    if conversation_id:
        conversation = AgentConversation.objects.filter(
            id=conversation_id,
            match=match,
            is_active=True,
        ).first()
        if conversation:
            return conversation

    title = f"{match.title} Agent Chat"
    if innings:
        title = f"{match.title} - Innings {innings.innings_number} Agent Chat"

    return AgentConversation.objects.create(
        match=match,
        innings=innings,
        created_by=user if getattr(user, "is_authenticated", False) else None,
        title=title,
    )


def get_recent_memory(conversation: AgentConversation, limit: int = 12) -> List[Dict[str, Any]]:
    messages = conversation.messages.order_by("-created_at")[:limit]
    messages = reversed(list(messages))

    return [
        {
            "role": msg.role,
            "content": msg.content,
            "metadata": msg.metadata,
            "created_at": msg.created_at.isoformat(),
        }
        for msg in messages
    ]


def save_agent_interaction(
    match: Match,
```

```
    innings: Optional[Innings],
    user_message: str,
    assistant_answer: str,
    metadata=None,
    user=None,
    conversation_id: Optional[int] = None,
):
    conversation = get_or_create_conversation(
        match=match,
        innings=innings,
        user=user,
        conversation_id=conversation_id,
    )

    AgentMessage.objects.create(
        conversation=conversation,
        role="user",
        content=user_message,
        metadata={},
    )

    AgentMessage.objects.create(
        conversation=conversation,
        role="assistant",
        content=assistant_answer,
        metadata=metadata or {},
    )

    return {"saved": True, "conversation_id": conversation.id}


def save_tool_message(conversation: AgentConversation, tool_name: str, payload: Dict[str, Any]):
    return AgentMessage.objects.create(
        conversation=conversation,
        role="tool",
        content=tool_name,
        metadata=payload,
    )
```

6. Context Builder

The context builder is the bridge between Django models and the LLM. It should not dump the entire database. It should create a compact, stable JSON object describing only what the agent needs: score, innings, current players, recent balls, and active banners.

```
# matches/ai_agent/context_builder.py

def build_match_context(match: Match, innings: Optional[Innings] = None) -> Dict[str, Any]:
    if innings is None:
        innings = (
            match.innings
            .select_related("match", "batting_team", "bowling_team")
            .order_by("-innings_number")
```

```
        .first()
    )

if not innings:
    return {
        "match_id": match.id,
        "match_name": str(match),
        "has_innings": False,
        "message": "No innings available yet.",
    }

scoring_state = _get_scoring_state(innings)
summary = _innings_summary(innings)

recent_balls = list(
    innings.ball_events
    .select_related("striker", "bowler", "dismissed_player")
    .order_by("-over_number", "-ball_number", "-id")[:12]
)
recent_balls.reverse()

active_infographic = cache.get(f"live_infographic_banner:innings:{innings.id}")
batting_players = list(innings.batting_team.players.all())
bowling_players = list(innings.bowling_team.players.all())

return {
    "match_id": match.id,
    "match_name": str(match),
    "has_innings": True,
    "innings_id": innings.id,
    "innings_number": innings.innings_number,
    "batting_team": innings.batting_team.name,
    "bowling_team": innings.bowling_team.name,
    "score": summary,
    "current_state": {
        "striker_id": scoring_state.striker_id,
        "non_striker_id": scoring_state.non_striker_id,
        "bowler_id": scoring_state.bowler_id,
    },
    "batting_players_available": [
        {"id": player.id, "name": player.name}
        for player in batting_players
    ],
    "bowling_players_available": [
        {"id": player.id, "name": player.name}
        for player in bowling_players
    ],
    "recent_balls": [
        {
            "over": ball.over_number,
            "ball": ball.ball_number,
            "striker": _safe_player_name(ball.striker),
            "bowler": _safe_player_name(ball.bowler),
            "runs_off_bat": ball.runs_off_bat,
            "extras": ball.extras,
            "wicket_fell": ball.wicket_fell,
```

```
            "dismissed_player": _safe_player_name(ball.dismissed_player),
            "shot_type": getattr(ball, "shot_type", ""),
            "shot_zone": getattr(ball, "shot_zone", ""),
        }
        for ball in recent_balls
    ],
    "active_infographics": [active_infographic] if active_infographic else [],
}
```

7. Prompt Layer and JSON Instruction

The prompt layer defines the agent identity, boundaries, available actions, and output schema. In this MVP, the model must return JSON only. Django parses that JSON and decides the tool call.

```
# matches/ai_agent/prompts.py

KHEL_AI_AGENT_SYSTEM_PROMPT = """
You are Khel AI Match Analyst Agent.

You are running inside the Khel AI Django live scoring webapp.

Your users are admins, scorers, mentors, or presenters. Viewers do not directly control you.

Your job:
1. Answer cricket analytics questions using only the provided match context and tool results.
2. Decide whether an external analytics tool is needed.
3. Create a live banner only when the admin explicitly asks for a banner/card/infographic/on-screen output.
4. Keep live-screen banner text short, broadcast-friendly, and readable.
5. Do not invent player stats. If data is missing, say what is missing.
6. Prefer current innings context over general cricket knowledge.
7. When the user asks for model/API outputs, route to the correct tool.
8. When the user asks for explanation, explain the metric in simple terms.
9. When the user asks for coaching/presentation language, create crisp commentary.
10. Never expose API keys, environment variables, raw stack traces, or internal settings.

Banner policy:
- Do not create a banner for normal analysis.
- Create a banner only if the admin says phrases like:
  "show on live match", "create banner", "display card",
  "put this on screen", "send to live page", "create infographic",
  "show this on stream", "make a live card".
- If a tool fails, you may create an agent_insight banner only if the user explicitly asked for a banner.

Safety and accuracy:
- Use match context first.
- Use tool results second.
- If context and tool result conflict, mention uncertainty.
- Do not pretend an API worked if it failed.
"""


AGENT_JSON_INSTRUCTION = """
Return ONLY valid JSON. No markdown. No explanation outside JSON.
```

Schema:

```
{
  "answer": "Response to show in the admin chat.",
  "tool_intent": "none | scorecard | batting | bowling | all_rounder | banner",
  "should_create_banner": false,
  "banner_request": {
    "metric_type": "batting_dashboard | consistency_index | pressure_performance | shot_risk_efficiency |
bowling_economy_deviation | wicket_probability_model | control_entropy_model | full_bowling_analysis |
weighted_contribution_index | correlation_analysis | performance_variance_model | full_all_rounder_analysis | agent_insight",
    "player_id": null,
    "display_area": "between_balls | between_overs | main_overlay | bottom_bar",
    "banner_title": "",
    "banner_text": ""
  },
  "reasoning_summary": "One short sentence explaining why this tool/metric was selected."
}
"""
```

8. Tool Layer and External API Integration

Tools are Python functions the agent is allowed to use. They should be deterministic, whitelisted, and wrapped in safe error handling. The LLM should never be allowed to provide arbitrary URLs or arbitrary Python code.

```
# matches/ai_agent/schemas.py
from dataclasses import dataclass, field
from typing import Any, Dict, List, Optional

@dataclass
class AgentToolResult:
    ok: bool
    tool_name: str
    data: Dict[str, Any] = field(default_factory=dict)
    error: Optional[str] = None

@dataclass
class AgentResponse:
    ok: bool
    answer: str
    tool_results: List[Dict[str, Any]]
    created_banner: bool = False
    created_card_id: Optional[int] = None
    error: Optional[str] = None


# matches/ai_agent/tools.py
import requests
from typing import Dict, Any, Optional
from django.conf import settings
```

```
def _post_json(url: str, payload: Dict[str, Any], timeout: Optional[int] = None) -> Dict[str, Any]:
    response = requests.post(
        url,
        json=payload,
        timeout=timeout or getattr(settings, "EXTERNAL_ANALYTICS_API_TIMEOUT", 25),
    )
    response.raise_for_status()
    return response.json()
```

Example: local scorecard tool

```
def get_current_scorecard(match: Match, innings: Optional[Innings]) -> AgentToolResult:
    if not innings:
        return AgentToolResult(False, "get_current_scorecard", error="No innings found.")

    from matches.services import innings_summary

    return AgentToolResult(
        ok=True,
        tool_name="get_current_scorecard",
        data=innings_summary(innings),
    )
```

Example: Student 1 API tool

```
def call_student1_consistency_index(innings: Innings, player: Player) -> AgentToolResult:
    try:
        payload = build_student1_sprint2_payload(innings, player)
        url = f"{settings.STUDENT1_SPRINT2_API_BASE_URL}/api/v1/batting/consistency-index"
        data = _post_json(url, payload)

        return AgentToolResult(
            ok=True,
            tool_name="call_student1_consistency_index",
            data={
                "payload_sent": payload,
                "api_response": data,
            },
        )
    except Exception as exc:
        return AgentToolResult(False, "call_student1_consistency_index", error=str(exc))
```

9. Tool Registry

A registry prevents agent.py from becoming a long chain of if/else statements. Every metric name maps to a Python function, a player role, and a description.

```
# matches/ai_agent/registry.py
from matches.ai_agent import tools

TOOL_REGISTRY = {
    "batting_dashboard": {
```

```
    "function": tools.call_student1_batting_dashboard,
    "role": "batter",
    "description": "Full batting dashboard for a batter.",
},
"consistency_index": {
    "function": tools.call_student1_consistency_index,
    "role": "batter",
    "description": "Batting consistency and stability.",
},
"pressure_performance": {
    "function": tools.call_student1_pressure_performance,
    "role": "batter",
    "description": "How a batter performs under pressure.",
},
"shot_risk_efficiency": {
    "function": tools.call_student1_shot_risk_efficiency,
    "role": "batter",
    "description": "Shot risk versus reward.",
},
"bowling_economy_deviation": {
    "function": tools.call_student2_bowling_economy_tool,
    "role": "bowler",
    "description": "Whether a bowler is expensive or economical.",
},
"wicket_probability_model": {
    "function": tools.call_student2_wicket_probability_tool,
    "role": "bowler",
    "description": "Wicket-taking probability.",
},
"control_entropy_model": {
    "function": tools.call_student2_control_entropy_tool,
    "role": "bowler",
    "description": "Bowler control, discipline, and variation.",
},
"full_bowling_analysis": {
    "function": tools.call_student2_full_bowling_analysis_tool,
    "role": "bowler",
    "description": "Complete bowling intelligence report.",
},
"weighted_contribution_index": {
    "function": tools.call_student3_weighted_contribution_tool,
    "role": "any",
    "description": "Weighted contribution across batting, bowling, and fielding.",
},
"correlation_analysis": {
    "function": tools.call_student3_correlation_analysis_tool,
    "role": "any",
    "description": "Relationship between batting and bowling performance.",
},
"performance_variance_model": {
    "function": tools.call_student3_performance_variance_tool,
    "role": "any",
    "description": "Reliability and variance across roles.",
},
"full_all_rounder_analysis": {
    "function": tools.call_student3_full_all_rounder_analysis_tool,
```

```
        "role": "any",
        "description": "Complete all-rounder report.",
    },
}


def get_tool(metric_type: str):
    return TOOL_REGISTRY.get(metric_type)
```

10. Agent Orchestration Code

agent.py is the orchestrator. It should do only orchestration: load context, load memory, ask the model, parse JSON, validate the tool request, execute whitelisted tools, optionally create a banner, save memory, and return JSON to the frontend.

```
# matches/ai_agent/agent.py - abbreviated production structure
import json
from typing import Dict, Any, Optional, List

from django.conf import settings
from openai import OpenAI

from matches.models import Match, Innings, Player
from matches.ai_agent.context_builder import build_match_context
from matches.ai_agent.prompts import KHEL_AI_AGENT_SYSTEM_PROMPT, AGENT_JSON_INSTRUCTION
from matches.ai_agent.tools import get_current_scorecard, create_agent_banner, create_api_result_banner
from matches.ai_agent.memory import get_or_create_conversation, get_recent_memory, save_agent_interaction
from matches.ai_agent.registry import get_tool
from matches.ai_agent.validators import clean_banner_request


def _safe_json_loads(text: str) -> Dict[str, Any]:
    try:
        return json.loads(text)
    except Exception:
        start = text.find("{")
        end = text.rfind("}")
        if start != -1 and end != -1 and end > start:
            try:
                return json.loads(text[start:end + 1])
            except Exception:
                pass

    return {
        "answer": text,
        "tool_intent": "none",
        "should_create_banner": False,
        "banner_request": {},
        "reasoning_summary": "Model did not return valid JSON.",
    }


def _as_bool(value: Any) -> bool:
```

```
    if isinstance(value, bool):
        return value
    if isinstance(value, str):
        return value.strip().lower() in {"true", "1", "yes"}
    return bool(value)


def _run_metric_tool(metric_type: str, innings: Innings, player: Player):
    tool_spec = get_tool(metric_type)
    if not tool_spec:
        tool_spec = get_tool("batting_dashboard")
    return tool_spec["function"](innings, player)


def run_khel_ai_agent(
    match: Match,
    message: str,
    innings: Optional[Innings] = None,
    allow_create_banner: bool = True,
    user=None,
    conversation_id: Optional[int] = None,
) -> Dict[str, Any]:
    if innings is None:
        innings = (
            match.innings
            .select_related("match", "batting_team", "bowling_team")
            .order_by("-innings_number")
            .first()
        )

    context = build_match_context(match, innings)

    if not settings.OPENAI_API_KEY:
        return {
            "ok": False,
            "answer": "OPENAI_API_KEY is missing. Add it to your environment first.",
            "tool_results": [],
            "created_banner": False,
            "conversation_id": None,
        }

    conversation = get_or_create_conversation(
        match=match,
        innings=innings,
        user=user,
        conversation_id=conversation_id,
    )
    recent_memory = get_recent_memory(conversation, limit=12)

    prompt = f"""
{KHEL_AI_AGENT_SYSTEM_PROMPT}

{AGENT_JSON_INSTRUCTION}

MATCH CONTEXT:
{json.dumps(context, default=str, indent=2)}
```

```
RECENT CONVERSATION MEMORY:
{json.dumps(recent_memory, default=str, indent=2)}

ADMIN MESSAGE:
{message}
"""

    client = OpenAI(api_key=settings.OPENAI_API_KEY)
    response = client.responses.create(
        model=getattr(settings, "KHEL_AI_AGENT_MODEL", "gpt-5.4-nano"),
        input=prompt,
    )

    parsed = _safe_json_loads(response.output_text)
    answer = parsed.get("answer", "No answer generated.")
    should_create_banner = _as_bool(parsed.get("should_create_banner"))
    banner_request = clean_banner_request(parsed.get("banner_request") or {})

    tool_results: List[Dict[str, Any]] = []
    created_banner = False

    if innings:
        score_tool = get_current_scorecard(match, innings)
        tool_results.append(score_tool.__dict__)

    # Banner/tool execution block goes here. Keep all tool calls whitelisted.

    memory_result = save_agent_interaction(
        match=match,
        innings=innings,
        user_message=message,
        assistant_answer=answer,
        metadata={
            "tool_results": tool_results,
            "created_banner": created_banner,
            "raw_model_output": parsed,
        },
        user=user,
        conversation_id=conversation.id,
    )

    return {
        "ok": True,
        "answer": answer,
        "created_banner": created_banner,
        "created_card_id": None,
        "conversation_id": memory_result.get("conversation_id", conversation.id),
        "tool_results": tool_results,
    }
```

11. Django View and URL Integration

The view receives the admin message from the browser, resolves match and innings, calls the agent runner, and returns JSON.

```
# matches/views.py
import json
from django.http import JsonResponse
from django.shortcuts import get_object_or_404
from django.views.decorators.http import require_POST
from django.contrib.auth.decorators import login_required

from matches.models import Match
from matches.ai_agent.agent import run_khel_ai_agent


@login_required
@require_POST
def khel_ai_agent_api(request, match_id):
    match = get_object_or_404(Match, pk=match_id)

    try:
        body = json.loads(request.body.decode("utf-8"))
    except Exception:
        body = request.POST

    message = body.get("message", "").strip()
    innings_id = body.get("innings_id")
    conversation_id = body.get("conversation_id")

    if not message:
        return JsonResponse({
            "ok": False,
            "error": "Message is required.",
        }, status=400)

    innings = None
    if innings_id:
        innings = match.innings.filter(id=innings_id).first()

    result = run_khel_ai_agent(
        match=match,
        innings=innings,
        message=message,
        allow_create_banner=True,
        user=request.user,
        conversation_id=conversation_id,
    )

    return JsonResponse(result)


# matches/urls.py
path(
    "api/agent/<int:match_id>/",
```

```
    views.khel_ai_agent_api,
    name="khel_ai_agent_api",
),
```

12. Frontend Admin Chat Integration

Add an admin chat panel to innings_scoring.html. The chat sends the admin message, innings ID, and conversation ID to the Django agent endpoint. The response is displayed in the scoring page.

```
<section class="agent-panel">
  <h3>Khel AI Agent</h3>

  <div id="agent-messages" class="agent-messages"></div>

  <form id="agent-form">
    {% csrf_token %}
    <input
      type="text"
      id="agent-input"
      placeholder="Ask: Who is under pressure? Show wicket probability banner..."
      autocomplete="off"
    />
    <button type="submit">Ask Agent</button>
  </form>
</section>

<script>
let agentConversationId = null;

document.getElementById("agent-form").addEventListener("submit", async function (event) {
  event.preventDefault();

  const input = document.getElementById("agent-input");
  const message = input.value.trim();
  if (!message) return;

  const messages = document.getElementById("agent-messages");
  messages.innerHTML += `<div class="msg user"><b>You:</b> ${message}</div>`;
  input.value = "";

  const response = await fetch("{% url 'khel_ai_agent_api' innings.match.id %}", {
    method: "POST",
    headers: {
      "Content-Type": "application/json",
      "X-CSRFToken": document.querySelector("[name=csrfmiddlewaretoken]").value,
    },
    body: JSON.stringify({
      message: message,
      innings_id: "{{ innings.id }}",
      conversation_id: agentConversationId,
    }),
  });
```

```
  const data = await response.json();

  if (data.conversation_id) {
    agentConversationId = data.conversation_id;
  }

  if (data.ok) {
    messages.innerHTML += `<div class="msg assistant"><b>Khel AI:</b> ${data.answer}</div>`;
    if (data.created_banner) {
      messages.innerHTML += `<div class="msg system">Live banner created.</div>`;
    }
  } else {
    messages.innerHTML += `<div class="msg error"><b>Error:</b> ${data.error || data.answer}</div>`;
  }

  messages.scrollTop = messages.scrollHeight;
});
</script>


.agent-panel {
  margin-top: 24px;
  padding: 16px;
  border-radius: 16px;
  background: #101827;
  color: #fff;
}
.agent-messages {
  max-height: 260px;
  overflow-y: auto;
  padding: 12px;
  background: rgba(255,255,255,0.06);
  border-radius: 12px;
  margin-bottom: 12px;
}
.msg { margin-bottom: 10px; line-height: 1.4; }
.msg.user { color: #dbeafe; }
.msg.assistant { color: #dcfce7; }
.msg.error { color: #fecaca; }
.msg.system { color: #fde68a; }
#agent-form { display: flex; gap: 8px; }
#agent-input { flex: 1; padding: 10px 12px; border-radius: 10px; border: none; }
#agent-form button { padding: 10px 16px; border-radius: 10px; border: none; font-weight: 700; cursor: pointer; }
```

13. Live Banner Pipeline

The live banner pipeline uses Django cache. The agent creates a temporary banner. The live page polls the analytics API. The analytics API returns the active banner until the cache TTL expires.

```
Agent creates banner
        ↓
create_temporary_live_banner()
        ↓
```

```
cache.set("live_infographic_banner:innings:{innings.id}", banner_payload, timeout=20)
        ↓
live_analytics_api reads the same cache key
        ↓
frontend polls /api/live/<match_id>/analytics/
        ↓
live_match.html shows the banner overlay
```

```
# matches/ai_agent/tools.py

def create_temporary_live_banner(
    innings: Innings,
    player: Player,
    metric_type: str,
    display_area: str,
    card_data: Dict[str, Any],
) -> AgentToolResult:
    banner_payload = {
        "id": f"temp-banner-{innings.id}",
        "player_name": player.name,
        "metric_type": metric_type,
        "metric_label": metric_type.replace("_", " ").title(),
        "display_area": display_area,
        "card_data": card_data,
    }

    cache.set(
        f"live_infographic_banner:innings:{innings.id}",
        banner_payload,
        timeout=getattr(settings, "LIVE_INFOGRAPHIC_TTL_SECONDS", 20),
    )

    return AgentToolResult(
        ok=True,
        tool_name="create_temporary_live_banner",
        data={
            "created_banner": True,
            "card_id": None,
            "banner": banner_payload,
        },
    )
```

Production cache recommendation

For local development, LocMemCache is acceptable. For Render or multi-worker production, use Redis via django-redis so all workers share the same live banner state.

14. Adding Sprint 3 PKL Model APIs as Tools

Sprint 3 APIs should be treated exactly like Student 1/2/3 analytics APIs. The model should be trained once, exported as a PKL with feature_columns and metrics, hosted behind a FastAPI/Django endpoint, and called by the Django agent as a tool. Do not retrain the model on every request.

```
# .env
PLAYER_RUNS_MODEL_API_URL=https://khel-ai-sprint3-player-runs.onrender.com/predict
PLAYER_RUNS_MODEL_API_TIMEOUT=90


# settings.py
PLAYER_RUNS_MODEL_API_URL = os.environ.get("PLAYER_RUNS_MODEL_API_URL", "")
PLAYER_RUNS_MODEL_API_TIMEOUT = int(os.environ.get("PLAYER_RUNS_MODEL_API_TIMEOUT", "90"))


def build_player_runs_prediction_payload(innings, player):
    from matches.services import innings_summary

    summary = innings_summary(innings)
    recent_balls = list(
        innings.ball_events
        .filter(striker=player)
        .order_by("-over_number", "-ball_number", "-id")[:12]
    )

    return {
        "player_id": player.id,
        "player_name": player.name,
        "team": player.team.name,
        "innings_id": innings.id,
        "match_id": innings.match.id,
        "current_score": summary["total_runs"],
        "wickets": summary["wickets"],
        "overs": summary["overs"],
        "recent_balls": [
            {
                "runs_off_bat": ball.runs_off_bat,
                "extras": ball.extras,
                "shot_type": ball.shot_type,
                "shot_zone": ball.shot_zone,
                "wicket_fell": ball.wicket_fell,
            }
            for ball in recent_balls
        ],
    }


def call_player_runs_prediction_tool(innings: Innings, player: Player) -> AgentToolResult:
    try:
        if not settings.PLAYER_RUNS_MODEL_API_URL:
            return AgentToolResult(
                ok=False,
                tool_name="call_player_runs_prediction_tool",
                error="PLAYER_RUNS_MODEL_API_URL is not configured.",
```

```
            )

        payload = build_player_runs_prediction_payload(innings, player)

        data = _post_json(
            settings.PLAYER_RUNS_MODEL_API_URL,
            payload,
            timeout=getattr(settings, "PLAYER_RUNS_MODEL_API_TIMEOUT", 90),
        )

        return AgentToolResult(
            ok=True,
            tool_name="call_player_runs_prediction_tool",
            data={"payload_sent": payload, "api_response": data},
        )

    except Exception as exc:
        return AgentToolResult(
            ok=False,
            tool_name="call_player_runs_prediction_tool",
            error=str(exc),
        )


# registry.py
"player_runs_prediction": {
    "function": tools.call_player_runs_prediction_tool,
    "role": "batter",
    "description": "Predicts expected player runs using a Sprint 3 PKL model API.",
},
```

15. Guardrails, Validation, and Security

The most important rule: the LLM suggests intent, but Django enforces permissions. The LLM should not be able to choose arbitrary URLs, write arbitrary files, run code, delete data, or expose secrets.

```
# matches/ai_agent/validators.py
from matches.ai_agent.registry import TOOL_REGISTRY

ALLOWED_DISPLAY_AREAS = {
    "between_balls",
    "between_overs",
    "main_overlay",
    "bottom_bar",
}

ALLOWED_METRICS = set(TOOL_REGISTRY.keys()) | {"agent_insight"}


def clean_banner_request(raw):
    if not isinstance(raw, dict):
        raw = {}
```

```
    metric_type = raw.get("metric_type") or "agent_insight"
    if metric_type not in ALLOWED_METRICS:
        metric_type = "agent_insight"

    display_area = raw.get("display_area") or "between_balls"
    if display_area not in ALLOWED_DISPLAY_AREAS:
        display_area = "between_balls"

    player_id = raw.get("player_id")
    try:
        player_id = int(player_id) if player_id else None
    except Exception:
        player_id = None

    return {
        "metric_type": metric_type,
        "player_id": player_id,
        "display_area": display_area,
        "banner_title": str(raw.get("banner_title") or "")[:120],
        "banner_text": str(raw.get("banner_text") or "")[:300],
    }
```

- [ ] Never expose OPENAI_API_KEY in any JSON response.
- [ ] Never allow the user or LLM to submit arbitrary API URLs.
- [ ] Never allow arbitrary Python execution.
- [ ] Require POST for the agent endpoint.
- [ ] Use CSRF protection in browser requests.
- [ ] Add login_required or staff-only guard before production.
- [ ] Validate metric_type against TOOL_REGISTRY.
- [ ] Validate display_area against allowed display locations.
- [ ] Store tool outputs in AgentMessage metadata.
- [ ] Show friendly API timeout messages instead of raw stack traces.
- [ ] Keep viewer-facing banners short and non-sensitive.
- [ ] Move to Redis cache before multi-worker deployment.

16. Testing and Debugging

Django checks

```
python manage.py check
python manage.py makemigrations
python manage.py migrate
python manage.py runserver
```

Manual test flow

- [ ] Create teams in Django admin.
- [ ] Create players for both teams.
- [ ] Create a match.

- [ ] Create an innings.
- [ ] Add at least 6-12 ball events.
- [ ] Open the innings scoring page.
- [ ] Ask: What is the current score?
- [ ] Ask: Who is under pressure?
- [ ] Ask: Show a batting consistency banner for the striker.
- [ ] Open the live match page in another tab.
- [ ] Confirm the banner appears.
- [ ] Ask: Show wicket probability for the current bowler.
- [ ] Confirm the Student 2 API is called.
- [ ] Ask: Just explain the current match situation.
- [ ] Confirm no banner is created.

Curl test

```
curl -X POST http://127.0.0.1:8000/api/agent/1/ \
  -H "Content-Type: application/json" \
  -d "{\"message\":\"Show wicket probability banner for the current bowler\",\"innings_id\":1}"
```

Expected JSON response

```
{
  "ok": true,
  "answer": "...",
  "created_banner": true,
  "created_card_id": null,
  "conversation_id": 1,
  "tool_results": []
}
```

17. Deployment Checklist

- [ ] Set OPENAI_API_KEY in Render environment variables.
- [ ] Set KHEL_AI_AGENT_MODEL in Render environment variables.
- [ ] Set all Student API base URLs in environment variables.
- [ ] Set DEBUG=False in production.
- [ ] Set ALLOWED_HOSTS correctly for Render domain and custom domain.
- [ ] Use PostgreSQL instead of SQLite for production.
- [ ] Use Redis for cache if multiple workers or instances are used.
- [ ] Run migrations during deployment.
- [ ] Collect static files if static serving is used.
- [ ] Protect the agent endpoint with login_required or staff_member_required.
- [ ] Review CORS/CSRF settings if frontend and backend are on different domains.
- [ ] Add API timeout and fallback messages for sleeping Render services.
- [ ] Add logging for tool calls and failed model outputs.

Render docs: https://docs.render.com/

18. Teaching Explanation for Students

Use this explanation in class:

> **Simple definition**
>
> An AI chatbot gives text. An AI agent can read context, decide which tool is needed, call an API, interpret the result, and take an action such as creating a live banner.

```
Chatbot = answer text
Agent   = answer text + call tools + create banners + remember context
```

Five parts of the Khel AI agent

- Context: What is happening in the match?
- Prompt: What role and rules does the agent follow?
- Tools: What actions can the agent take?
- Memory: What has already been discussed?
- Response: What should the admin and live screen see?

19. Implementation Roadmap

- [ ] Step 1: Update requirements.txt.
- [ ] Step 2: Move secrets into .env and Render environment variables.
- [ ] Step 3: Add AgentConversation and AgentMessage models.
- [ ] Step 4: Run makemigrations and migrate.
- [ ] Step 5: Replace memory.py with database memory.
- [ ] Step 6: Add registry.py.
- [ ] Step 7: Add validators.py.
- [ ] Step 8: Upgrade prompts.py.
- [ ] Step 9: Upgrade context_builder.py to include batting and bowling players.
- [ ] Step 10: Refactor agent.py to use memory, registry, and validators.
- [ ] Step 11: Update khel_ai_agent_api to accept conversation_id and request.user.
- [ ] Step 12: Add admin chat UI to innings_scoring.html.
- [ ] Step 13: Confirm live banner cache pipeline works.
- [ ] Step 14: Add one Sprint 3 model API tool.
- [ ] Step 15: Test and document before adding the next model API.

20. Appendices: Code Snippets and Sample Prompts

Sample admin prompts

- What is the current score and match situation?
- Who is under the most pressure right now?
- Explain the last over in simple broadcast language.

- Show a batting consistency banner for the striker.
- Show wicket probability for the current bowler on the live screen.
- Create a main overlay card for the best all-rounder impact.
- Do not create a banner. Just explain whether the bowler has control.
- Which batter should be highlighted in the bottom bar?
- Create a short infographic for shot risk efficiency.

Available API endpoint map

Endpoint	Path	Purpose
Agent API	/api/agent/<match_id>/	POST admin message to Khel AI agent
Live analytics API	/api/live/<match_id>/analytics/	Returns analytics cards and active live banner
Scoreboard API	/api/matches/<match_id>/scoreboard/	Returns match scoreboard JSON
Bowler momentum proxy	/api/innings/<innings_id>/bowler/<player_id>/momentum-proxy/	Calls bowler momentum API
Student 2 economy proxy	/api/innings/<innings_id>/bowler/<player_id>/student2/economy-deviation/	Calls Student 2 economy deviation API
Student 2 wicket proxy	/api/innings/<innings_id>/bowler/<player_id>/student2/wicket-probability/	Calls Student 2 wicket probability API
Student 2 control proxy	/api/innings/<innings_id>/bowler/<player_id>/student2/control-entropy/	Calls Student 2 control entropy API
Student 3 contribution proxy	/api/innings/<innings_id>/player/<player_id>/student3/weighted-contribution/	Calls Student 3 weighted contribution API

Render URL inventory

These are the active URLs referenced in the current Khel AI MVP settings and sprint architecture. Keep them in .env/settings.py and do not hardcode them deep inside tool functions.

Bowler Momentum API - https://bowler-momentum-api-3.onrender.com/api/bowler-momentum/

Student 1 Sprint 1 - https://khel-ai-st1-sp1-1.onrender.com

Student 2 Sprint 1 - https://khel-ai-st2-sp1.onrender.com

Student 3 Sprint 1 - https://khel-ai-st3-sp1.onrender.com

Student 4 Sprint 1 - https://khel-ai-st4-sp1.onrender.com

Student 5 Sprint 1 - https://khel-ai-st5-sp1.onrender.com

Student 1 Sprint 2 - https://khel-ai-st1-sp2.onrender.com

Student 2 Sprint 2 - https://khel-ai-st2-sp2.onrender.com

Student 3 Sprint 2 - https://khel-ai-st3-sp2.onrender.com

Final principle

Build rule

Do not let the model directly control the system. Let the model choose intent. Let Django enforce what tools exist, what payloads are valid, what actions are allowed, and what gets displayed.

End of Khel AI Django AI Agent Build Guide