Futmatrix AI System

Product Technical Architecture – Version 2.0

Product Name

Futmatrix AI System

Vision

"Design is not just what it looks like and feels like. Design is how it works."

Futmatrix is not just a Discord bot. It is a hybrid Web + AI gaming platform designed to empower competitive players of EAFC25. The platform intelligently collects, processes, and interprets game data — including match stats, training progress, streaming activity, and behavioral metrics — and transforms these into meaningful insights, rankings, and token-based incentives.

---

1. System Overview

1.1 Purpose

To provide an integrated gaming experience where users:

- Compete in skill-matched Rivalizer PvP games
- Receive personalized coaching from an AI Coach
- Upload gameplay images/videos for analysis
- Track their performance in dashboards and weekly rankings
- Earn or lose tokens based on behavior and performance

All of this happens across a web platform and Discord, powered by AI agents and microservices.

1.2 Core Principles

- **Automation with Intention:** Every service is modular yet synchronized via queues.

- **Real-Time Feedback:** AI agents give near-instant responses via web and Discord.
- **Token-Driven Game Economy:** Incentives and penalties are applied via smart contracts.
- **Data Ownership:** Players maintain custody of their tokens; the platform is non-custodial.

---

2. Functional Components

2.1 Web Platform

- User authentication (via Supabase + Whop)
- Dashboard UI: metrics, training status, streaming history
- Ranking pages: "Week on Fire" and "Rivalizer Arena"
- Match suggestion screen (Rivalizer matchmaking interface)
- Upload module: video/image upload for AI Coach

2.2 Discord Bot

- Companion interface for match uploads, commands, and notifications
- Syncs with the platform (via webhooks and Supabase)

2.3 AI Agents

- **Coach Agent**: Evaluates performance and tracks training plan adherence
- **Rivalizer Agent**: Suggests matches, logs outcomes, triggers smart contract payouts

---

3. Microservices Architecture

Each component is containerized and communicates via RabbitMQ. Primary services:

3.1 `discord-listener-service`

- Listens to Discord image uploads
- Extracts metadata
- Publishes to `image_ingested` queue

3.2 `image-ocr-service`

- Consumes `image_ingested`
- Sends image to OpenAI Vision API
- Parses stats and publishes to `image_ocr_result`

3.3 `raw-data-processor-service`

- Converts raw OCR output into structured match metrics
- Detects game mode (friendly, rivals, rivalizer)
- Publishes to `data_ready_for_storage`

3.4 `database-controller-service`

- Writes structured data to Supabase tables:
  - `matches`
  - `processed_metrics`
  - `user_stats_summary`
- Ensures match type and coverage level are indexed

3.5 `agent-logger-service`

- Tracks interactions between users and AI agents (Coach, Rivalizer)

- Stores in `agent_interactions` table

3.6 `training-plan-service`

- Manages creation and tracking of training plans
- Checks weekly goals and issues token rewards or penalties

3.7 `ranking-engine`

- Computes weekly rankings:
    - `week_on_fire`
    - `rivalizer_arena`
- Runs weekly cronjob, stores in cache tables

---

4. Database Schema (Supabase)

Key tables:

- `users`: extended with `whop_id`, `subscription_status`
- `user_plans`
- `matches`: includes `match_type`, `is_ranked`, `data_coverage_level`
- `processed_metrics`
- `agent_interactions`: logs chat, uploads, match suggestions
- `training_plans`: tracks stake, checkpoints, compliance
- `penalties`: yellow/red card system
- `streaming_rewards`, `replay_uploads`

- `weekly_rankings`: cache for top 50 players (multiple modes)

---

5. Smart Contracts Overview

- **Token Contract** (BEP-20): Fixed supply
- **Match Escrow Contract**: Handles staking and winner payout
- **Reward Distributor**: Handles bonuses, streaming payouts
- **Penalty Contract**: Manages yellow/red card fines
- **Training Stake Contract**: Stakes and distributes rewards for training plans

All contracts are non-custodial and triggered by backend or oracle.

---

6. Deployment

- Each service is containerized via Docker
- Orchestrated using Docker Compose or Kubernetes
- Persistent queues via RabbitMQ
- Supabase for storage, auth, and views
- Optional Redis for cache/rankings

---

7. Extensibility

- Plug-and-play metrics architecture
- Easily introduce new game modes or rank types
- Tokenomics logic tied to backend events via modular reward pipeline

- All agent responses driven by stored views (e.g., `coach_user_view`)

---

8. AI Agents Layer (LangGraph)

The Coach and Rivalizer agents are implemented as LangGraph agents in Python.
Each agent is a stateful computation graph with memory access and tool-based orchestration.

8.1 Coach Agent (LangGraph)

- Input: user_id
- Tools:
  - Supabase DB Query Tool (fetch last 5 matches)
  - Metrics Analyzer Tool (compute avg efficiency)
  - Training Plan Tool (generate or update plan)
- Output: performance summary + updated training goal
- Logic:
  - If no recent match: suggest playing
  - If below performance threshold: adjust plan
  - If consistent progress: increase challenge + issue reward

8.2 Rivalizer Agent (LangGraph)

- Input: user_id
- Tools:
  - Supabase DB Tool (match history)
  - Matchmaking Tool (access to rivalizer_matchmaking_view)
  - Discord + Platform Integration Tool (sends match invite)

    - Smart Contract Trigger Tool (stake, payout)
- Output: suggested match + match booking
- Logic:
    - Filters eligible opponents
    - Picks 3 optimal based on skill delta + recency
    - Records user acceptance and schedules match

---

9. Final Note

This is not just a bot. It’s a player-owned competitive gaming economy. Designed with rigor, extensibility, and delight in mind.