```
{
"memo_type": "monolith_structure",
"source_file": "fortuna.py",
"part": 3,
"total_parts": 3,
"blocks": [
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "class",
"content": "class HotTipsTracker:\n \"\"\"Logs reported opportunities to a SQLite database.\"\"\"\n def __init__(self,
db_path: Optional[str] = None, config: Optional[Dict[str, Any]] = None):\n self.db = FortunaDB(db_path) if db_path else
FortunaDB()\n self.config = config or {}\n self.logger = structlog.get_logger(self.__class__.__name__)\n\n async def
log_tips(self, races: List[Race]):\n if not races:\n return\n\n await self.db.initialize()\n now = datetime.now(EASTERN)\n
report_date = to_storage_format(now)\n new_tips = []\n already_handled_soft_keys = set()\n\n # Future cutoff relaxed to allow
advance tips\n future_limit = now + timedelta(hours=24)\n\n for r in races:\n # Only store \"Best Bets\" (Goldmine, BET NOW,
or You Might Like)\n # These are marked in metadata by the analyzer.\n # FIX_09: Also include pure superfecta keybox plays\n
if not r.metadata.get('is_best_bet') and not r.metadata.get('is_goldmine') and not r.metadata.get('is_superfecta_key'):\n
continue\n\n # Trustworthiness Airlock Safeguard (Council of Superbrains Directive)\n active_runners = [run for run in
r.runners if not run.scratched]\n total_active = len(active_runners)\n\n # Ensure trustworthy odds exist before logging\n if
r.metadata.get('predicted_2nd_fav_odds') is None:\n continue\n\n if total_active > 0:\n trustworthy_count = sum(1 for run in
active_runners if run.metadata.get(\"odds_source_trustworthy\"))\n trust_ratio = trustworthy_count / total_active\n # Relaxed
to match SimplySuccessAnalyzer config (GPT5 alignment)\n # BUG-2 Fix: Align with expected config key\n min_trust =
self.config.get(\"analysis\", {}).get(\"simply_success_trust_min\", 0.25)\n if trust_ratio < min_trust:\n
self.logger.warning(\"Rejecting race with low trust_ratio for DB logging\", venue=r.venue, race=r.race_number,
trust_ratio=round(trust_ratio, 2), required=min_trust)\n continue\n\n st = r.start_time\n if isinstance(st, str):\n try: st =
from_storage_format(st.replace('Z', '+00:00'))\n except Exception: continue\n if st.tzinfo is None: st =
st.replace(tzinfo=EASTERN)\n\n # Reject races too far in the future\n # BUG-4 Fix: Expand timing gate to 60m to prevent
dropping late-detected goldmines\n if st > future_limit or st < now - timedelta(minutes=60):\n self.logger.debug(\"Rejecting
far-future or ancient race\", venue=r.venue, start_time=st)\n continue\n\n # BUG-12: Secondary soft-key dedup guard\n soft_key
= f\"{get_canonical_venue(r.venue)}|{r.race_number}|{st.strftime('%y%m%d')}\"\n if soft_key in already_handled_soft_keys:\n
self.logger.debug(\"Skipping duplicate play (soft key match)\", soft_key=soft_key)\n continue\n
already_handled_soft_keys.add(soft_key)\n\n is_goldmine = r.metadata.get('is_goldmine', False)\n gap12 =
r.metadata.get('1Gap2', 0.0)\n\n tip_data = {\n \"report_date\": report_date,\n \"race_id\": r.id,\n \"venue\": r.venue,\n
\"race_number\": r.race_number,\n \"start_time\": to_storage_format(r.start_time) if isinstance(r.start_time, datetime) else
str(r.start_time),\n \"is_goldmine\": is_goldmine,\n \"source\": r.source,\n \"1Gap2\": gap12,\n \"discipline\":
r.discipline,\n \"top_five\": r.top_five_numbers,\n \"selection_number\": r.metadata.get('selection_number'),\n
\"selection_name\": r.metadata.get('selection_name'),\n \"predicted_2nd_fav_odds\":
r.metadata.get('predicted_2nd_fav_odds'),\n \"field_size\": total_active,\n \"market_depth\":
r.metadata.get('market_depth'),\n \"place_prob\": r.metadata.get('place_prob'),\n \"predicted_ev\":
r.metadata.get('predicted_ev'),\n \"race_type\": getattr(r, 'race_type', None),\n \"is_handicap\": getattr(r, 'is_handicap',
None),\n \"condition_modifier\": r.metadata.get('condition_modifier'),\n \"qualification_grade\":
r.metadata.get('qualification_grade'),\n \"composite_score\": r.metadata.get('composite_score'),\n \"is_best_bet\":
r.metadata.get('is_best_bet', False),\n \"is_superfecta_key\": r.metadata.get('is_superfecta_key', False),\n
\"superfecta_key_number\": r.metadata.get('superfecta_key_number'),\n \"superfecta_key_name\":
r.metadata.get('superfecta_key_name')\n }\n new_tips.append(tip_data)\n\n try:\n # Cap the batch size to avoid performance
degradation (GPT5 Improvement)\n if len(new_tips) > 100:\n self.logger.info(\"Capping large tips batch\",
original_count=len(new_tips), capped_at=100)\n new_tips = new_tips[:100]\n\n await self.db.log_tips(new_tips)\n
self.logger.info(\"Hot tips processed\", count=len(new_tips))\n except Exception as e:\n self.logger.error(\"Failed to log hot
tips\", error=str(e))\n",
"name": "HotTipsTracker"
},
{
"type": "miscellaneous",
"content": "\n\n# ----------------------------------------\n# MONITOR LOGIC\n#
----------------------------------------\n#!/usr/bin/env python3\n"
},
{
"type": "docstring",
"content": "\"\"\"\nFortuna Favorite-to-Place Betting Monitor\n=========================================\n\nThis script
monitors racing data from multiple adapters and identifies\nbetting opportunities based on:\n1. Second favorite odds >= 4.0
decimal\n2. Races under 120 minutes to post (MTP)\n3. Superfecta availability preferred\n\nUsage:\n python
favorite_to_place_monitor.py [--date YYMMDD] [--refresh-interval 30]\n\"\"\"\n"
},
{
"type": "miscellaneous",
"content": "\n@dataclass\n"
},
{
"type": "class",
"content": "class RaceSummary:\n \"\"\"Summary of a single race for display.\"\"\"\n discipline: str # T/H/G\n track: str\n
race_number: int\n field_size: int\n superfecta_offered: bool\n adapter: str\n start_time: datetime\n mtp: Optional[int] =
None # Minutes to post\n second_fav_odds: Optional[float] = None\n second_fav_name: Optional[str] = None\n selection_number:
Optional[int] = None\n favorite_odds: Optional[float] = None\n favorite_name: Optional[str] = None\n top_five_numbers:
Optional[str] = None\n gap12: float = 0.0\n is_goldmine: bool = False\n is_best_bet: bool = False\n is_superfecta_key: bool =
False\n superfecta_key_number: Optional[int] = None\n superfecta_key_name: Optional[str] = None\n superfecta_box_numbers:
List[int] = Field(default_factory=list)\n\n def to_dict(self) -> dict:\n \"\"\"Convert to dictionary for JSON
serialization.\"\"\"\n return {\n \"discipline\": self.discipline,\n \"track\": self.track,\n \"race_number\":
self.race_number,\n \"field_size\": self.field_size,\n \"superfecta_offered\": self.superfecta_offered,\n \"adapter\":
self.adapter,\n \"start_time\": to_storage_format(self.start_time),\n \"mtp\": self.mtp,\n \"second_fav_odds\":
self.second_fav_odds,\n \"second_fav_name\": self.second_fav_name,\n \"selection_number\": self.selection_number,\n
\"favorite_odds\": self.favorite_odds,\n \"favorite_name\": self.favorite_name,\n \"top_five_numbers\":
self.top_five_numbers,\n \"gap12\": self.gap12,\n \"is_goldmine\": self.is_goldmine,\n \"is_best_bet\": self.is_best_bet,\n
\"is_superfecta_key\": self.is_superfecta_key,\n \"superfecta_key_number\": self.superfecta_key_number,\n
\"superfecta_key_name\": self.superfecta_key_name,\n \"superfecta_box_numbers\": self.superfecta_box_numbers\n }\n",
"name": "RaceSummary"
},
{
"type": "miscellaneous",
"content": "\n\n@lru_cache(maxsize=1)\n"
},
{
"type": "function",
"content": "def get_discovery_adapter_classes() -> List[Type[BaseAdapterV3]]:\n \"\"\"Recursively discovers all discovery
adapter classes (cached for performance - GPT5 Improvement).\"\"\"\n def get_all_subclasses(cls):\n return
set(cls.__subclasses__()).union(\n [s for c in cls.__subclasses__() for s in get_all_subclasses(c)]\n )\n\n return [\n c for c
in get_all_subclasses(BaseAdapterV3)\n if not getattr(c, \"__abstractmethods__\", None)\n and getattr(c, \"ADAPTER_TYPE\",
\"discovery\") == \"discovery\"\n and not getattr(c, \"DECOMMISSIONED\", False)\n ]\n",
"name": "get_discovery_adapter_classes"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "class",
"content": "class FavoriteToPlaceMonitor:\n \"\"\"Monitor for favorite-to-place betting opportunities.\"\"\"\n\n def
__init__(self, target_dates: Optional[List[str]] = None, refresh_interval: int = 30, config: Optional[Dict] = None):\n
\"\"\"\n Initialize monitor.\n\n Args:\n target_dates: Dates to fetch races for (YYMMDD), defaults to today + tomorrow\n
refresh_interval: Seconds between refreshes for BET NOW list\n \"\"\"\n if target_dates:\n self.target_dates = target_dates\n
else:\n today = datetime.now(EASTERN)\n tomorrow = today + timedelta(days=1)\n self.target_dates =
[today.strftime(DATE_FORMAT), tomorrow.strftime(DATE_FORMAT)]\n\n self.refresh_interval = refresh_interval\n self.config =
config or {}\n self.all_races: List[RaceSummary] = []\n self.adapters: List = []\n self.logger =
structlog.get_logger(self.__class__.__name__)\n self.tracker = HotTipsTracker(config=self.config)\n\n async def
initialize_adapters(self, adapter_names: Optional[List[str]] = None):\n \"\"\"Initialize all adapters, optionally filtered by
name.\"\"\"\n all_discovery_classes = get_discovery_adapter_classes()\n\n classes_to_init = all_discovery_classes\n if
adapter_names:\n classes_to_init = [c for c in all_discovery_classes if c.__name__ in adapter_names or getattr(c,
\"SOURCE_NAME\", \"\") in adapter_names]\n\n self.logger.info(\"Initializing adapters\", count=len(classes_to_init))\n\n # Get
adapter-specific configs from global config (GPT5 Improvement)\n adapter_configs = self.config.get(\"adapters\", {})\n\n for
adapter_class in classes_to_init:\n try:\n name = adapter_class.SOURCE_NAME if hasattr(adapter_class, \"SOURCE_NAME\") else
adapter_class.__name__\n specific_config = adapter_configs.get(name, {}).copy() # Use copy to avoid shared mutation\n # Merge
with basic region config\n specific_config.update({\"region\": self.config.get(\"region\")})\n adapter =
adapter_class(config=specific_config)\n self.adapters.append(adapter)\n self.logger.debug(\"Adapter initialized\",
adapter=adapter_class.__name__)\n except Exception as e:\n self.logger.error(\"Adapter initialization failed\",
adapter=adapter_class.__name__, error=str(e))\n\n self.logger.info(\"Adapters initialization complete\",
initialized=len(self.adapters))\n\n async def fetch_all_races(self) -> List[Tuple[Race, str]]:\n \"\"\"Fetch races from all
adapters.\"\"\"\n self.logger.info(\"Fetching races\", dates=self.target_dates)\n\n all_races_with_adapters = []\n\n # Run
fetches in parallel for speed\n async def fetch_one(adapter, date_str):\n name = adapter.__class__.__name__\n try:\n races =
await adapter.get_races(date_str)\n self.logger.info(\"Fetch complete\", adapter=name, date=date_str, count=len(races))\n
return [(r, name) for r in races]\n except Exception as e:\n self.logger.error(\"Fetch failed\", adapter=name, date=date_str,
error=str(e))\n return []\n\n fetch_tasks = []\n for d in self.target_dates:\n for a in self.adapters:\n
fetch_tasks.append(fetch_one(a, d))\n\n results = await asyncio.gather(*fetch_tasks)\n for r_list in results:\n
all_races_with_adapters.extend(r_list)\n\n self.logger.info(\"Total races fetched\", total=len(all_races_with_adapters))\n
return all_races_with_adapters\n\n def _get_discipline_code(self, race: Race) -> str:\n \"\"\"Get discipline code
(T/H/G).\"\"\"\n if not race.discipline:\n return \"T\"\n\n d = race.discipline.lower()\n if \"harness\" in d or
\"standardbred\" in d: return \"H\"\n if \"greyhound\" in d or \"dog\" in d: return \"G\"\n return \"T\"\n\n def
_calculate_field_size(self, race: Race) -> int:\n \"\"\"Calculate active field size.\"\"\"\n return len([r for r in
race.runners if not r.scratched])\n\n def _has_superfecta(self, race: Race) -> bool:\n \"\"\"Check if race offers
Superfecta.\"\"\"\n ab = race.available_bets or []\n # Support metadata fallback if field not populated\n if not ab and
hasattr(race, 'metadata'):\n ab = race.metadata.get('available_bets', [])\n return \"Superfecta\" in ab\n\n def
_get_top_runners(self, race: Race, limit: int = 5) -> List[Runner]:\n \"\"\"Get top runners by odds, sorted lowest
first.\"\"\"\n # Get active runners with valid odds\n r_with_odds = []\n for r in race.runners:\n if r.scratched:\n continue\n
# Refresh odds to avoid stale metadata in continuous monitor mode\n wo = _get_best_win_odds(r)\n if wo is not None and wo >
1.0:\n # Update runner object with fresh odds for downstream summaries\n r.win_odds = float(wo)\n # Store the Decimal odds
directly for sorting to avoid conversion\n r_with_odds.append((r, wo))\n\n if not r_with_odds:\n return []\n\n # Sort by odds
(lowest first)\n sorted_r = sorted(r_with_odds, key=lambda x: x[1])\n return [x[0] for x in sorted_r[:limit]]\n\n def
_calculate_mtp(self, start_time: Optional[datetime]) -> int:\n \"\"\"Calculate minutes to post. Returns -9999 if start_time is
None.\"\"\"\n if not start_time: return -9999\n now = now_eastern()\n # Use ensure_eastern to handle naive or other timezones
correctly\n st = ensure_eastern(start_time)\n delta = st - now\n return int(delta.total_seconds() / 60)\n\n def
_get_top_n_runners(self, race: Race, n: int = 5) -> str:\n \"\"\"Get top N runners by win odds.\"\"\"\n top_runners =
self._get_top_runners(race, limit=n)\n return \", \".join([str(r.number) if r.number is not None else \"?\" for r in
top_runners])\n\n def _create_race_summary(self, race: Race, adapter_name: str) -> RaceSummary:\n \"\"\"Create a RaceSummary
from a Race object.\"\"\"\n top_runners = self._get_top_runners(race, limit=5)\n favorite = top_runners[0] if len(top_runners)
>= 1 else None\n second_fav = top_runners[1] if len(top_runners) >= 2 else None\n\n gap12 = 0.0\n if favorite and second_fav
and favorite.win_odds and second_fav.win_odds:\n gap12 = round((second_fav.win_odds - favorite.win_odds) / favorite.win_odds,
2)\n\n return RaceSummary(\n discipline=self._get_discipline_code(race),\n track=normalize_venue_name(race.venue),\n
race_number=race.race_number,\n field_size=self._calculate_field_size(race),\n
superfecta_offered=self._has_superfecta(race),\n adapter=adapter_name,\n start_time=race.start_time,\n
mtp=self._calculate_mtp(race.start_time),\n second_fav_odds=second_fav.win_odds if second_fav else None,\n
second_fav_name=second_fav.name if second_fav else None,\n selection_number=second_fav.number if second_fav else None,\n
favorite_odds=favorite.win_odds if favorite else None,\n favorite_name=favorite.name if favorite else None,\n
top_five_numbers=self._get_top_n_runners(race, 5),\n gap12=gap12,\n is_goldmine=race.metadata.get('is_goldmine', False),\n
is_best_bet=race.metadata.get('is_best_bet', False),\n is_superfecta_key=race.metadata.get('is_superfecta_key', False),\n
superfecta_key_number=race.metadata.get('superfecta_key_number'),\n
superfecta_key_name=race.metadata.get('superfecta_key_name'),\n
superfecta_box_numbers=race.metadata.get('superfecta_box_numbers', [])\n )\n\n async def build_race_summaries(self,
races_with_adapters: List[Tuple[Race, str]], window_hours: Optional[int] = 12):\n \"\"\"Build and deduplicate summary list,
with optional time window filtering.\"\"\"\n race_map = {}\n now = datetime.now(EASTERN)\n cutoff = now +
timedelta(hours=window_hours) if window_hours else None\n\n adapter_scores = await self.tracker.db.get_adapter_scores(days=30)
if hasattr(self.tracker, 'db') else {}\n\n for race, adapter_name in races_with_adapters:\n try:\n # Time window filtering\n
st = race.start_time\n if not st: continue # Guard against None start_time\n if st.tzinfo is None: st =
st.replace(tzinfo=EASTERN)\n\n # Time window filtering removed to ensure all unique races are counted\n\n summary =
self._create_race_summary(race, adapter_name)\n # Stable key: Canonical Venue + Race Number + Date + Discipline\n
canonical_venue = get_canonical_venue(summary.track)\n date_str = summary.start_time.strftime('%y%m%d') if summary.start_time
else \"Unknown\"\n key = f\"{canonical_venue}|{summary.race_number}|{date_str}|{summary.discipline}\"\n\n if key not in
race_map:\n race_map[key] = summary\n else:\n existing = race_map[key]\n incoming_odds = summary.second_fav_odds or 0.0\n
existing_odds = existing.second_fav_odds or 0.0\n if incoming_odds > existing_odds:\n summary.superfecta_offered =
summary.superfecta_offered or existing.superfecta_offered\n race_map[key] = summary\n elif incoming_odds == existing_odds:\n
incoming_score = adapter_scores.get(summary.adapter, 0)\n existing_score = adapter_scores.get(existing.adapter, 0)\n if
incoming_score > existing_score:\n summary.superfecta_offered = summary.superfecta_offered or existing.superfecta_offered\n
race_map[key] = summary\n elif summary.superfecta_offered and not existing.superfecta_offered:\n existing.superfecta_offered =
True\n else:\n if summary.superfecta_offered and not existing.superfecta_offered:\n existing.superfecta_offered = True\n
except Exception: pass\n\n unique_summaries = list(race_map.values())\n self.all_races = sorted(unique_summaries, key=lambda
x: x.start_time)\n\n # GPT5 Improvement: Keep all races within window for analysis, not just one per track.\n # Window
broadened to 18 hours (News Mode)\n timing_window_summaries = []\n now = datetime.now(EASTERN)\n for summary in
unique_summaries:\n st = summary.start_time\n if st.tzinfo is None: st = st.replace(tzinfo=EASTERN)\n\n # Calculate Minutes to
Post\n diff = st - now\n mtp = diff.total_seconds() / 60\n\n # Timing window limited to 8 hours to ensure yield is
audit-able\n if -45 < mtp <= 480: # 8 hours\n timing_window_summaries.append(summary)\n\n self.golden_zone_races =
timing_window_summaries\n if not self.golden_zone_races:\n self.logger.warning(\"\ud83d\udd2d Monitor found 0 races in the
timing window (-45m to 8h)\", total_unique=len(unique_summaries))\n\n def print_full_list(self):\n \"\"\"Log all fetched
races.\"\"\"\n lines = [\n \"=\" * 120,\n \"FULL RACE LIST\".center(120),\n \"=\" * 120,\n f\"{'DISC':<5} {'TRACK':<25}
{'R#':<4} {'FIELD':<6} {'SUPER':<6} {'ADAPTER':<25} {'START TIME':<20}\",\n \"-\" * 120\n ]\n for r in sorted(self.all_races,
key=lambda x: (x.discipline, x.track, x.race_number)):\n superfecta = \"Yes\" if r.superfecta_offered else \"No\"\n # Display
time in Eastern with ET suffix\n st = r.start_time.strftime(\"%y%m%dT%H:%M ET\") if r.start_time else \"Unknown\"\n
lines.append(f\"{r.discipline:<5} {r.track[:24]:<25} {r.race_number:<4} {r.field_size:<6} {superfecta:<6} {r.adapter[:24]:<25}
{st:<20}\")\n lines.append(\"-\" * 120)\n lines.append(f\"Total races: {len(self.all_races)}\")\n
self.logger.info(\"\\n\".join(lines))\n\n def get_bet_now_races(self) -> List[RaceSummary]:\n \"\"\"Get races meeting BET NOW
criteria (GPT5 Alignment).\"\"\"\n # Configurable thresholds\n ana_config = self.config.get(\"analysis\", {})\n min_odds =
ana_config.get(\"bet_now_min_odds\", 4.0)\n max_field = ana_config.get(\"max_field_size\", 11)\n min_gap =
ana_config.get(\"min_gap\", 0.25)\n mtp_limit = ana_config.get(\"bet_now_mtp_limit\", 120)\n\n bet_now = [\n r for r in
self.golden_zone_races\n if r.mtp is not None and -10 < r.mtp <= mtp_limit\n and r.second_fav_odds is not None and
r.second_fav_odds >= min_odds\n and r.field_size <= max_field\n and r.gap12 >= min_gap\n ]\n # Sort by Superfecta desc, then
MTP asc\n bet_now.sort(key=lambda r: (not r.superfecta_offered, r.mtp))\n return bet_now[:15] # Cap to prevent overwhelming
output\n\n def get_you_might_like_races(self, bet_now_races: Optional[List[RaceSummary]] = None) -> List[RaceSummary]:\n
\"\"\"Get 'You Might Like' races with relaxed criteria (GPT5 Optimized).\"\"\"\n # Configurable thresholds\n ana_config =
self.config.get(\"analysis\", {})\n min_odds = ana_config.get(\"yml_min_odds\", 3.0)\n max_field =
ana_config.get(\"max_field_size\", 11)\n min_gap = ana_config.get(\"min_gap\", 0.25)\n mtp_limit =
ana_config.get(\"yml_mtp_limit\", 240)\n\n if bet_now_races is None:\n bet_now_races = self.get_bet_now_races()\n bet_now_keys
= {(r.track, r.race_number) for r in bet_now_races}\n yml = [\n r for r in self.golden_zone_races\n if r.mtp is not None and
-10 < r.mtp <= mtp_limit\n and r.second_fav_odds is not None and r.second_fav_odds >= min_odds\n and r.field_size <=
max_field\n and r.gap12 >= min_gap\n and (r.track, r.race_number) not in bet_now_keys\n ]\n # Sort by MTP asc\n
yml.sort(key=lambda r: r.mtp)\n return yml[:5] # Limit to top 5 recommendations\n\n async def print_bet_now_list(self):\n
\"\"\"Log filtered BET NOW list and recent audited goldmine results.\"\"\"\n bet_now = self.get_bet_now_races()\n lines = [\n
\"=\" * 140,\n \"\ud83c\udfaf BET NOW - FAVORITE TO PLACE OPPORTUNITIES\".center(140),\n \"=\" * 140,\n f\"Updated:
{datetime.now(EASTERN).strftime(' %H:%M:%S')} ET\",\n \"Criteria: -10 < MTP <= 120 minutes AND 2nd Favorite Odds >= 4.0\",\n
\"-\" * 140\n ]\n if not bet_now:\n lines.append(\"\u23f3 No races currently meet BET NOW criteria.\")\n yml =
self.get_you_might_like_races(bet_now_races=bet_now)\n if yml:\n lines.extend([\n \"=\" * 160,\n \"\ud83c\udf1f YOU MIGHT LIKE
- NEAR-MISS OPPORTUNITIES\".center(160),\n \"=\" * 160,\n f\"{'SUPER':<6} {'MTP':<5} {'DISC':<5} {'TRACK':<20} {'R#':<4}
{'FIELD':<6} {'ODDS':<20} {'TOP 5':<20}\",\n \"-\" * 160\n ])\n for r in yml:\n sup = \"\u2705\" if r.superfecta_offered else
\"\u274c\"\n fo = f\"{r.favorite_odds:.2f}\" if r.favorite_odds else \"N/A\"\n so = f\"{r.second_fav_odds:.2f}\" if
r.second_fav_odds else \"N/A\"\n top5 = r.top_five_numbers or \"N/A\"\n # Leading zero alignment\n m_str = f\"{r.mtp:02d}\" if
0 <= r.mtp < 10 else str(r.mtp)\n lines.append(f\"{sup:<6} {m_str:<5} {r.discipline:<5} {r.track[:19]:<20} {r.race_number:<4}
{r.field_size:<6} ~ {fo}, {so:<15} [{top5}]\")\n lines.append(\"-\" * 160)\n self.logger.info(\"\\n\".join(lines))\n
return\n\n lines.extend([\n f\"{'SUPER':<6} {'MTP':<5} {'DISC':<5} {'TRACK':<20} {'R#':<4} {'FIELD':<6} {'ODDS':<20} {'TOP
5':<20}\",\n \"-\" * 160\n ])\n for r in bet_now:\n sup = \"\u2705\" if r.superfecta_offered else \"\u274c\"\n fo =
f\"{r.favorite_odds:.2f}\" if r.favorite_odds else \"N/A\"\n so = f\"{r.second_fav_odds:.2f}\" if r.second_fav_odds else
\"N/A\"\n top5 = r.top_five_numbers or \"N/A\"\n m_str = f\"{r.mtp:02d}\" if 0 <= r.mtp < 10 else str(r.mtp)\n
lines.append(f\"{sup:<6} {m_str:<5} {r.discipline:<5} {r.track[:19]:<20} {r.race_number:<4} {r.field_size:<6} ~ {fo}, {so:<15}
[{top5}]\")\n lines.extend([\"-\" * 160, f\"Total opportunities: {len(bet_now)}\"])\n
self.logger.info(\"\\n\".join(lines))\n\n # Include recent audited results to provide proof of system performance\n history =
await self.tracker.db.get_recent_audited_goldmines(limit=10)\n if history:\n historical_report =
generate_historical_goldmine_report(history)\n self.logger.info(historical_report)\n\n def save_to_json(self, filename: str =
\"race_data.json\"):\n \"\"\"Export to JSON.\"\"\"\n bn = self.get_bet_now_races()\n yml =
self.get_you_might_like_races(bet_now_races=bn)\n\n target_file = get_writable_path(filename)\n alert_file =
get_writable_path(\"monitor_empty.alert\")\n\n if not bn:\n self.logger.warning(\"\ud83d\udd2d Monitor found 0 BET NOW
opportunities\", total_checked=len(self.golden_zone_races))\n # Structured telemetry for monitoring\n
structlog.get_logger(\"FortunaTelemetry\").warning(\"empty_bet_now_list\", golden_zone_count=len(self.golden_zone_races))\n #
Create an indicator file for downstream monitoring (GPT5 Improvement)\n try:\n
alert_file.write_text(to_storage_format(datetime.now(EASTERN)))\n except Exception: pass\n else:\n # Clear alert if it
exists\n try:\n if alert_file.exists(): alert_file.unlink()\n except Exception: pass\n\n data = {\n \"generated_at\":
to_storage_format(datetime.now(EASTERN)),\n \"target_dates\": self.target_dates,\n \"total_races\": len(self.all_races),\n
\"bet_now_count\": len(bn),\n \"you_might_like_count\": len(yml),\n \"all_races\": [r.to_dict() for r in self.all_races],\n
\"bet_now_races\": [r.to_dict() for r in bn],\n \"you_might_like_races\": [r.to_dict() for r in yml],\n }\n try:\n # Ensure
parent directory exists (GPT5 Improvement)\n target_file.parent.mkdir(parents=True, exist_ok=True)\n with open(target_file,
'w', encoding='utf-8') as f:\n json.dump(data, f, indent=2)\n except Exception as e:\n
self.logger.error(\"failed_saving_race_data\", path=str(target_file), error=str(e))\n\n # Persistent history log\n
self._append_to_history(bn + yml)\n\n def _append_to_history(self, races: List[RaceSummary]):\n \"\"\"Append races to
persistent history for future result matching.\"\"\"\n if not races: return\n history_file =
get_writable_path(\"prediction_history.jsonl\")\n\n # Improvement 04: Rotation logic\n try:\n if history_file.exists() and
history_file.stat().st_size > 10 * 1024 * 1024: # 10MB\n backup = history_file.with_suffix(\".jsonl.1\")\n
history_file.replace(backup)\n self.logger.info(\"Rotated prediction history file\")\n except Exception: pass\n\n timestamp =
to_storage_format(datetime.now(EASTERN))\n try:\n with open(history_file, 'a', encoding='utf-8') as f:\n for r in races:\n
record = r.to_dict()\n record[\"logged_at\"] = timestamp\n f.write(json.dumps(record) + \"\\n\")\n except Exception as e:\n
```

```
self.logger.error(\"History logging failed\", error=str(e))\n\n async def run_once(self, loaded_races: Optional[List[Race]] =
None, adapter_names: Optional[List[str]] = None):\n try:\n if loaded_races is not None:\n self.logger.info(\"Using loaded
races\", count=len(loaded_races))\n # Map to (Race, AdapterName) tuple expected by build_race_summaries\n raw = [(r, r.source)
for r in loaded_races]\n else:\n await self.initialize_adapters(adapter_names=adapter_names)\n raw = await
self.fetch_all_races()\n\n await self.build_race_summaries(raw, window_hours=12) # Use 12h window for monitor\n
self.print_full_list()\n await self.print_bet_now_list()\n for r in self.all_races:\n r.mtp =
self._calculate_mtp(r.start_time)\n self.save_to_json()\n finally:\n for a in self.adapters: await a.shutdown()\n await
GlobalResourceManager.cleanup()\n\n async def run_continuous(self):\n await self.initialize_adapters()\n raw = await
self.fetch_all_races()\n await self.build_race_summaries(raw, window_hours=12)\n self.print_full_list()\n try:\n for _ in
range(1000): # Iteration limit to prevent potential hangs\n for r in self.all_races: r.mtp =
self._calculate_mtp(r.start_time)\n await self.print_bet_now_list()\n self.save_to_json()\n await
asyncio.sleep(self.refresh_interval)\n except KeyboardInterrupt:\n self.logger.info(\"Stopped by user\")\n except
asyncio.CancelledError:\n self.logger.info(\"Monitor task cancelled\")\n finally:\n for a in self.adapters: await
a.shutdown()\n await GlobalResourceManager.cleanup()\n",
"name": "FavoriteToPlaceMonitor"
},
{
"type": "miscellaneous",
"content": "\n\n\n\n\n# ----------------------------------------\n# EXPANDED ADAPTERS\n#
----------------------------------------\n# python_service/adapters/oddschecker_adapter.py\n\n\n\n\n\n"
},
{
"type": "class",
"content": "class OddscheckerAdapter(BrowserHeadersMixin, DebugMixin, BaseAdapterV3):\n \"\"\"Adapter for scraping horse
racing odds from Oddschecker, migrated to BaseAdapterV3.\"\"\"\n\n SOURCE_NAME = \"Oddschecker\"\n BASE_URL =
\"https://www.oddschecker.com\"\n\n def __init__(self, config=None):\n super().__init__(source_name=self.SOURCE_NAME,
base_url=self.BASE_URL, config=config)\n\n def _configure_fetch_strategy(self) -> FetchStrategy:\n # Oddschecker is heavily
protected by Cloudflare; Playwright with high timeout and network idle\n return FetchStrategy(\n
primary_engine=BrowserEngine.PLAYWRIGHT,\n enable_js=True,\n stealth_mode=\"camouflage\",\n timeout=120,\n network_idle=True\n
)\n\n async def make_request(self, method: str, url: str, **kwargs: Any) -> Any:\n # Playwright doesn't use impersonate but
SmartFetcher handles it now\n return await super().make_request(method, url, **kwargs)\n\n def _get_headers(self) -> dict:\n
return self._get_browser_headers(host=\"www.oddschecker.com\")\n\n async def _fetch_data(self, date: str) -> Optional[dict]:\n
\"\"\"\n Fetches the raw HTML for all race pages for a given date. This involves a multi-level fetch.\n \"\"\"\n sem =
asyncio.Semaphore(3)\n dt = parse_date_string(date)\n date_iso = dt.strftime(\"%Y-%m-%d\")\n index_url =
f\"/horse-racing/{date_iso}\"\n index_response = await self.make_request(\"GET\", index_url, headers=self._get_headers())\n if
not index_response or not index_response.text:\n self.logger.warning(\"Failed to fetch Oddschecker index page\",
url=index_url)\n return None\n\n self._save_debug_html(index_response.text, f\"oddschecker_index_{date}\")\n\n parser =
HTMLParser(index_response.text)\n # Find all links to individual race pages\n metadata = []\n\n try:\n target_date =
parse_date_string(date).date()\n except Exception:\n target_date = datetime.now(EASTERN).date()\n\n site_tz =
ZoneInfo(\"Europe/London\")\n now_site = datetime.now(site_tz)\n\n # Group by track to pick \"next\" race\n track_map =
defaultdict(list)\n\n # Broaden selectors for race links\n for selector in [\"a.race-time-link[href]\",
\"a[href*='/horse-racing/'][href*='/20']\", \".rf__link\"]:\n for a in parser.css(selector):\n href =
a.attributes.get(\"href\")\n if href and not href.endswith(\"/horse-racing\"):\n # Ensure absolute URL\n full_url = href if
href.startswith(\"http\") else f\"{self.BASE_URL}{href}\"\n # Extract track from URL if possible, or use parent\n # URL
usually /horse-racing/venue/date/time\n parts = full_url.split(\"/\")\n if len(parts) >= 6:\n track = parts[4]\n txt =
node_text(a) # Time is often in text\n track_map[track].append({\"url\": full_url, \"time_txt\": txt})\n\n for track, races in
track_map.items():\n for r in races:\n if re.match(r\"\\d{1,2}:\\d{2}\", r[\"time_txt\"]):\n try:\n rt =
datetime.strptime(r[\"time_txt\"], \"%H:%M\").replace(\n year=target_date.year, month=target_date.month, day=target_date.day,
tzinfo=site_tz\n )\n # Broaden window to capture multiple races\n diff = (rt - now_site).total_seconds() / 60\n if not (-45 <
diff <= 1080):\n continue\n\n metadata.append(r[\"url\"])\n except Exception: pass\n\n if not metadata:\n
self.logger.warning(\"No metadata found\", context=\"Oddschecker Index Parsing\", url=index_url)\n
self.metrics.record_parse_warning()\n return None\n\n async def fetch_single_html(url_path: str):\n async with sem:\n # Small
delay to avoid ban\n await asyncio.sleep(0.5 + random.random() * 0.5)\n response = await self.make_request(\"GET\", url_path,
headers=self._get_headers())\n return response.text if response else \"\"\n\n tasks = [fetch_single_html(link) for link in
metadata]\n html_pages = await asyncio.gather(*tasks)\n return {\"pages\": html_pages, \"date\": date}\n\n def
_parse_races(self, raw_data: Any) -> List[Race]:\n \"\"\"Parses a list of raw HTML strings from different races into Race
objects.\"\"\"\n if not raw_data or not raw_data.get(\"pages\"):\n return []\n\n try:\n race_date =
parse_date_string(raw_data[\"date\"]).date()\n except ValueError:\n self.logger.error(\n \"Invalid date format provided to
OddscheckerAdapter\",\n date=raw_data.get(\"date\"),\n )\n return []\n\n all_races = []\n for html in raw_data[\"pages\"]:\n
if not html:\n continue\n try:\n parser = HTMLParser(html)\n race = self._parse_race_page(parser, race_date)\n if race:\n
all_races.append(race)\n except (AttributeError, IndexError, ValueError):\n self.logger.warning(\n \"Error parsing a race from
Oddschecker, skipping race.\",\n exc_info=True,\n )\n continue\n return all_races\n\n def _parse_race_page(self, parser:
HTMLParser, race_date) -> Optional[Race]:\n track_name_node = parser.css_first(\"h1.meeting-name\")\n if not
track_name_node:\n return None\n track_name = node_text(track_name_node)\n\n race_time_node =
parser.css_first(\"span.race-time\")\n if not race_time_node:\n return None\n race_time_str = node_text(race_time_node)\n\n #
Heuristic to find race number from navigation\n active_link = parser.css_first(\"a.race-time-link.active\")\n race_number =
0\n if active_link:\n all_links = parser.css(\"a.race-time-link\")\n try:\n for i, link in enumerate(all_links):\n if
link.html == active_link.html:\n race_number = i + 1\n break\n except Exception:\n pass\n\n start_time =
datetime.combine(race_date, datetime.strptime(race_time_str, \"%H:%M\").time())\n runners = [runner for row in
parser.css(\"tr.race-card-row\") if (runner := self._parse_runner_row(row))]\n\n if not runners:\n return None\n\n # BUG-6
Fix: Use canonical venue for race ID\n venue_key = get_canonical_venue(track_name).lower().replace(' ', '')\n return Race(\n
id=f\"oc_{venue_key}_{start_time.strftime('%y%m%d')}_r{race_number}\",\n venue=track_name,\n race_number=race_number,\n
start_time=start_time,\n runners=runners,\n source=self.source_name,\n )\n\n def _parse_runner_row(self, row: Node) ->
Optional[Runner]:\n try:\n name_node = row.css_first(\"span.selection-name\")\n if not name_node:\n return None\n name =
node_text(name_node)\n\n odds_node = row.css_first(\"span.bet-button-odds-desktop, span.best-price\")\n if not odds_node:\n
return None\n odds_str = node_text(odds_node)\n\n number_node = row.css_first(\"td.runner-number\")\n number = 0\n if
number_node:\n num_txt = \"\".join(filter(str.isdigit, node_text(number_node)))\n if num_txt:\n number = int(num_txt)\n\n if
not name or not odds_str:\n return None\n\n win_odds = parse_odds_to_decimal(odds_str)\n\n # Advanced heuristic fallback\n if
win_odds is None:\n win_odds = SmartOddsExtractor.extract_from_node(row)\n\n odds_dict = {}\n if odds_data :=
create_odds_data(self.source_name, win_odds):\n odds_dict[self.source_name] = odds_data\n\n return Runner(number=number,
name=name, odds=odds_dict)\n except (AttributeError, ValueError):\n self.logger.warning(\"Failed to parse a runner on
Oddschecker, skipping runner.\")\n return None\n",
"name": "OddscheckerAdapter"
},
{
```

```
"type": "miscellaneous",
"content": "\n# python_service/adapters/timeform_adapter.py\n\n\n\n\n\n"
},
{
"type": "class",
"content": "class TimeformAdapter(JSONParsingMixin, BrowserHeadersMixin, DebugMixin, BaseAdapterV3):\n \"\"\"\n Adapter for
timeform.com, migrated to BaseAdapterV3 and standardized on selectolax.\n \"\"\"\n\n SOURCE_NAME = \"Timeform\"\n BASE_URL =
\"https://www.timeform.com\"\n\n def __init__(self, config=None):\n super().__init__(source_name=self.SOURCE_NAME,
base_url=self.BASE_URL, config=config)\n\n def _configure_fetch_strategy(self) -> FetchStrategy:\n # Timeform often blocks
basic requests; Playwright is robust\n return FetchStrategy(\n primary_engine=BrowserEngine.PLAYWRIGHT,\n enable_js=True,\n
stealth_mode=\"camouflage\",\n timeout=90,\n network_idle=True\n )\n\n def _get_headers(self) -> dict:\n headers =
self._get_browser_headers(host=\"www.timeform.com\")\n headers.update({\n \"Accept\":
\"text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8\",\n \"Accept-Language\":
\"en-US,en;q=0.9\",\n })\n return headers\n\n async def _fetch_data(self, date: str) -> Optional[dict]:\n \"\"\"\n Fetches the
raw HTML for all race pages for a given date.\n \"\"\"\n sem = asyncio.Semaphore(5)\n dt = parse_date_string(date)\n date_iso
= dt.strftime(\"%Y-%m-%d\")\n index_url = f\"/horse-racing/racecards/{date_iso}\"\n index_response = await
self.make_request(\"GET\", index_url, headers=self._get_headers())\n if not index_response or not index_response.text:\n
self.logger.warning(\"Failed to fetch Timeform index page\", url=index_url)\n return None\n\n
self._save_debug_snapshot(index_response.text, f\"timeform_index_{date}\")\n\n parser = HTMLParser(index_response.text)\n #
Updated selector for race links\n try:\n target_date = parse_date_string(date).date()\n except Exception:\n target_date =
datetime.now(EASTERN).date()\n\n site_tz = ZoneInfo(\"Europe/London\")\n now_site = datetime.now(site_tz)\n\n track_map =
defaultdict(list)\n # Broaden selectors for Timeform race links\n for selector in [\"a[href*='/racecards/']\", \".rf__link\",
\"a.rf-meeting-race__time\", \".rp-meetingItem__race__time\"]:\n for a in parser.css(selector):\n href =
a.attributes.get(\"href\")\n if href and \"/racecards/\" in href and not href.endswith(\"/racecards\"):\n # URL usually:
/horse-racing/racecards/venue/date/time/...\n # or: /racecards/venue/date/time\n parts = href.split(\"/\")\n # Handle both
relative and absolute-ish paths\n track = \"unknown\"\n for i, p in enumerate(parts):\n if p == \"racecards\" and i + 1 <
len(parts):\n track = parts[i+1]\n break\n\n txt = node_text(a)\n track_map[track].append({\"url\": href, \"time_txt\":
txt})\n\n links = []\n for track, races in track_map.items():\n for r in races:\n # Timeform often uses HH:MM in text\n
time_match = re.search(r\"(\\d{1,2}:\\d{2})\", r[\"time_txt\"])\n if time_match:\n try:\n rt =
datetime.strptime(time_match.group(1), \"%H:%M\").replace(\n year=target_date.year, month=target_date.month,
day=target_date.day, tzinfo=site_tz\n )\n # Broaden window to capture multiple races\n diff = (rt - now_site).total_seconds()
/ 60\n if not (-45 < diff <= 1080):\n continue\n\n full_url = r[\"url\"] if r[\"url\"].startswith(\"http\") else
f\"{self.BASE_URL}{r['url']}\"\n links.append(full_url)\n except Exception: pass\n\n if not links:\n self.logger.warning(\"No
metadata found\", context=\"Timeform Index Parsing\", url=index_url)\n self.metrics.record_parse_warning()\n return None\n\n
async def fetch_single_html(url_path: str):\n async with sem:\n await asyncio.sleep(0.5)\n response = await
self.make_request(\"GET\", url_path, headers=self._get_headers())\n return (url_path, response.text) if response else
(url_path, \"\")\n\n self.logger.info(f\"Found {len(links)} race links on Timeform\")\n tasks = [fetch_single_html(link) for
link in links]\n results = await asyncio.gather(*tasks)\n return {\"pages\": [r for r in results if r[1]], \"date\": date}\n\n
def _parse_races(self, raw_data: Any) -> List[Race]:\n \"\"\"Parses a list of raw HTML strings into Race objects.\"\"\"\n if
not raw_data or not raw_data.get(\"pages\"):\n return []\n\n try:\n race_date = parse_date_string(raw_data[\"date\"]).date()\n
except ValueError:\n self.logger.error(\"Invalid date format\", date=raw_data.get(\"date\"))\n return []\n\n all_races = []\n
for url_path, html_content in raw_data[\"pages\"]:\n if not html_content:\n continue\n try:\n parser =
HTMLParser(html_content)\n\n # Extract via JSON-LD if possible\n venue = \"\"\n start_time = None\n is_handicap = None\n
scripts = self._parse_all_jsons_from_scripts(parser, 'script[type=\"application/ld+json\"]', context=\"Betfair Index\")\n for
data in scripts:\n if data.get(\"@type\") == \"Event\":\n venue = normalize_venue_name(data.get(\"location\",
{}).get(\"name\", \"\"))\n if sd := data.get(\"startDate\"):\n # 2026-01-28T14:32:00\n start_time =
from_storage_format(sd.split('+')[0])\n break\n\n title_node = parser.css_first(\"title\")\n if title_node:\n title_text =
node_text(title_node)\n if \"HANDICAP\" in title_text.upper():\n is_handicap = True\n\n # BUG-17: Prefer URL-based time
extraction to avoid shared card-start-time issues\n url_time = None\n time_match = re.search(r'/(\\d{4})/?$',
url_path.split('?')[0])\n if time_match:\n try:\n url_time = datetime.combine(\n race_date,\n
datetime.strptime(time_match.group(1), \"%H%M\").time()\n )\n except Exception: pass\n\n if url_time:\n start_time =
url_time\n\n if not venue:\n # Fallback to title\n if title_node:\n # 14:32 DUNDALK | Races 28 January 2026 ...\n match =
re.search(r'(\\d{1,2}:\\d{2})\\s+([^|]+)', title_text)\n if match:\n time_str = match.group(1)\n venue =
normalize_venue_name(match.group(2).strip())\n if not start_time:\n start_time = datetime.combine(race_date,
datetime.strptime(time_str, \"%H:%M\").time())\n\n if not venue or not start_time:\n continue\n\n # Betting Forecast Parsing\n
forecast_map = {}\n verdict_section = parser.css_first(\"section.rp-verdict\")\n if verdict_section:\n forecast_text =
clean_text(node_text(verdict_section))\n if \"Betting Forecast :\" in forecast_text:\n # \"Betting Forecast : 15/8 2.87 Spring
Is Here, 3/1 4 This Guy, ...\"\n after_forecast = forecast_text.split(\"Betting Forecast :\")[1]\n # Split by comma\n parts =
after_forecast.split(',')\n for part in parts:\n # Match odds and then name\n # Odds can be fractional space decimal\n m =
re.search(r'(\\d+/\\d+|EVENS)\\s+([\\d\\.]+)?\\s*(.+)', part.strip())\n if m:\n odds_str = m.group(1)\n name =
clean_text(m.group(3))\n forecast_map[name.lower()] = odds_str\n\n # Runners\n runners = []\n # Use tbody as the main
container for each runner\n for row in parser.css('tbody.rp-horse-row'):\n if runner := self._parse_runner(row,
forecast_map):\n runners.append(runner)\n\n if not runners:\n continue\n\n # Race number from URL or sequence\n race_number =
0\n num_match = re.search(r'/(\\d+)/([^/]+)$', url_path)\n # .../1432/207/1/view... -> the '1' is the race number\n url_parts
= url_path.split('/')\n if len(url_parts) >= 10:\n try: race_number = int(url_parts[9])\n except Exception: pass\n\n race =
Race(\n id=f\"tf_{venue.lower().replace(' ', '')}_{start_time:%y%m%d}_R{race_number}\",\n venue=venue,\n
race_number=race_number,\n start_time=start_time,\n runners=runners,\n is_handicap=is_handicap,\n source=self.source_name,\n
)\n all_races.append(race)\n except Exception as e:\n self.logger.warning(f\"Error parsing Timeform race: {e}\")\n continue\n
return all_races\n\n def _parse_runner(self, row: Node, forecast_map: dict = None) -> Optional[Runner]:\n \"\"\"Parses a
single runner from a table row node.\"\"\"\n try:\n name_node = row.css_first(\"a.rp-horse\") or
row.css_first(\"a.rp-horseTable_horse-name\")\n if not name_node:\n return None\n name = clean_text(node_text(name_node))\n\n
number = 0\n num_attr = row.attributes.get(\"data-entrynumber\")\n if num_attr:\n try:\n val = int(num_attr)\n if val <= 40:
number = val\n except Exception:\n pass\n\n if not number:\n num_node = row.css_first(\".rp-entry-number\") or
row.css_first(\"span.rp-horseTable_horse-number\")\n if num_node:\n num_text = clean_text(node_text(num_node)).strip(\"()\")\n
num_match = re.search(r\"\\d+\", num_text)\n if num_match:\n val = int(num_match.group())\n if val <= 40: number = val\n\n
win_odds = None\n odds_source = None\n if forecast_map:\n win_odds = parse_odds_to_decimal(forecast_map.get(name.lower()))\n
if win_odds is not None:\n odds_source = \"morning_line\"\n\n # Try to find live odds button if available (old selector)\n if
not win_odds:\n odds_tag = row.css_first(\"button.rp-bet-placer-btn__odds\")\n if odds_tag:\n win_odds =
parse_odds_to_decimal(clean_text(node_text(odds_tag)))\n if win_odds is not None:\n odds_source = \"extracted\"\n\n # Advanced
heuristic fallback\n if win_odds is None:\n win_odds = SmartOddsExtractor.extract_from_node(row)\n if win_odds is not None:\n
odds_source = \"smart_extractor\"\n\n odds_data = {}\n if odds_val := create_odds_data(self.source_name, win_odds):\n
odds_data[self.source_name] = odds_val\n\n return Runner(number=number, name=name, win_odds=win_odds, odds=odds_data,
odds_source=odds_source)\n except (AttributeError, ValueError, TypeError):\n return None\n",
"name": "TimeformAdapter"
},
{
"type": "miscellaneous",
"content": "\n# python_service/adapters/racingpost_adapter.py\n\n\n\n\n"
},
{
"type": "class",
"content": "class RacingPostAdapter(BrowserHeadersMixin, DebugMixin, BaseAdapterV3):\n \"\"\"\n Adapter for scraping Racing
Post racecards, migrated to BaseAdapterV3.\n \"\"\"\n\n SOURCE_NAME = \"RacingPost\"\n BASE_URL =
\"https://www.racingpost.com\"\n\n def __init__(self, config=None):\n super().__init__(source_name=self.SOURCE_NAME,
base_url=self.BASE_URL, config=config)\n self.headers.update({\n \"Accept\":
\"text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.
\"Accept-Language\": \"en-US,en;q=0.9\",\n })\n\n def _configure_fetch_strategy(self) -> FetchStrategy:\n # Optimized for
speed: CURL_CFFI is much faster than Playwright for large batches of racecards.\n return FetchStrategy(\n
primary_engine=BrowserEngine.CURL_CFFI,\n enable_js=False,\n stealth_mode=\"camouflage\",\n timeout=60,\n
block_resources=True\n )\n\n def _get_headers(self) -> dict:\n headers =
self._get_browser_headers(host=\"www.racingpost.com\")\n headers.update({\n 'Accept':
'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8',\n 'Accept-Language': 'en-US,en;q=0.5',\n 'Accept-Encoding':
'gzip, deflate, br',\n })\n return headers\n\n async def _fetch_data(self, date: str) -> Any:\n \"\"\"\n Fetches the raw HTML
content for all races on a given date, including international.\n \"\"\"\n dt = parse_date_string(date)\n date_iso =
dt.strftime(\"%Y-%m-%d\")\n index_url = f\"/racecards/{date_iso}\"\n # RacingPost international URL sometimes varies\n
intl_urls = [\n f\"/racecards/international/{date_iso}\",\n f\"/racecards/{date_iso}/international\",\n
\"/racecards/international\"\n ]\n\n index_response = await self.make_request(\"GET\", index_url,
headers=self._get_headers())\n\n intl_response = None\n for url in intl_urls:\n resp = await self.make_request(\"GET\", url,
headers=self._get_headers())\n if resp and resp.status == 200:\n intl_response = resp\n break\n\n race_card_urls = []\n try:\n
target_date = parse_date_string(date).date()\n except Exception:\n target_date = datetime.now(EASTERN).date()\n\n site_tz =
ZoneInfo(\"Europe/London\")\n now_site = datetime.now(site_tz)\n\n if index_response and index_response.text:\n
self._save_debug_html(index_response.text, f\"racingpost_index_{date}\")\n index_parser = HTMLParser(index_response.text)\n\n
# Broaden window to capture multiple races\n meetings = index_parser.css('.rp-raceCourse__panel') or
index_parser.css('.RC-meetingItem') or index_parser.css('.rp-meetingItem') or index_parser.css('.RC-courseCards')\n for
meeting in meetings:\n # Broaden a tag selectors to catch new Racing Post structures\n for link in
meeting.css('a[data-test-selector^=\"RC-meetingItem__link_race\"], a.rp-raceCourse__panel__race__time,
a.rp-meetingItem__race__time, a.RC-meetingItem__race__time, a.RC-meetingItem__link, a[href*=\"/racecards/\"]'):\n href =
link.attributes.get(\"href\", \"\")\n if not href or \"/results/\" in href:\n continue\n\n txt = clean_text(node_text(link))\n
time_match = re.search(r\"(\\d{1,2}:\\d{2})\", txt)\n if time_match:\n try:\n time_str = time_match.group(1)\n tm =
datetime.strptime(time_str, \"%H:%M\")\n if tm.hour < 9:\n tm = tm.replace(hour=tm.hour + 12)\n\n rt = tm.replace(\n
year=target_date.year, month=target_date.month, day=target_date.day, tzinfo=site_tz\n )\n diff = (rt -
now_site).total_seconds() / 60\n if not (-45 < diff <= 1080):\n continue\n except Exception: pass\n\n
race_card_urls.append(href)\n\n elif index_response:\n self.logger.warning(\"Unexpected status\",
status=index_response.status, url=index_url)\n\n if intl_response and intl_response.text:\n
self._save_debug_html(intl_response.text, f\"racingpost_intl_index_{date}\")\n intl_parser =
HTMLParser(intl_response.text)\n\n meetings = intl_parser.css('.rp-raceCourse__panel') or intl_parser.css('.RC-meetingItem')
or intl_parser.css('.rp-meetingItem') or intl_parser.css('.RC-courseCards')\n for meeting in meetings:\n for link in
meeting.css('a[data-test-selector^=\"RC-meetingItem__link_race\"], a.rp-raceCourse__panel__race__time,
a.rp-meetingItem__race__time, a.RC-meetingItem__race__time, a.RC-meetingItem__link, a[href*=\"/racecards/\"]'):\n href =
link.attributes.get(\"href\", \"\")\n if not href or \"/results/\" in href:\n continue\n\n txt = clean_text(node_text(link))\n
time_match = re.search(r\"(\\d{1,2}:\\d{2})\", txt)\n if time_match:\n try:\n time_str = time_match.group(1)\n tm =
datetime.strptime(time_str, \"%H:%M\")\n if tm.hour < 9:\n tm = tm.replace(hour=tm.hour + 12)\n\n rt = tm.replace(\n
year=target_date.year, month=target_date.month, day=target_date.day, tzinfo=site_tz\n )\n diff = (rt -
now_site).total_seconds() / 60\n if not (-45 < diff <= 1080):\n continue\n except Exception: pass\n\n
race_card_urls.append(href)\n elif intl_response:\n self.logger.warning(\"Unexpected status\", status=intl_response.status,
url=intl_url)\n\n if not race_card_urls:\n self.logger.warning(\"Standard RacingPost link discovery failed, trying aggressive
fallback\", date=date)\n for resp in [index_response, intl_response]:\n if resp and resp.text:\n p = HTMLParser(resp.text)\n #
Even more aggressive: any link containing /racecards/ and a date-like pattern\n for a in p.css('a[href*=\"/racecards/\"]'):\n
href = a.attributes.get(\"href\", \"\")\n if re.search(r\"/\\d{4}-\\d{2}-\\d{2}/\", href) or re.search(r\"/\\d+/.*/\\d+/?$\",
href):\n race_card_urls.append(href)\n\n if not race_card_urls:\n self.logger.warning(\"Failed to fetch RacingPost racecard
links\", date=date)\n self.metrics.record_parse_warning()\n return None\n\n # Deduplicate URLs to avoid redundant fetching\n
race_card_urls = list(dict.fromkeys(race_card_urls))\n self.logger.info(\"Deduplicated RacingPost links\",
original=len(race_card_urls), unique=len(race_card_urls))\n\n async def fetch_single_html(url: str):\n response = await
self.make_request(\"GET\", url, headers=self._get_headers())\n return response.text if response else \"\"\n\n tasks =
[fetch_single_html(url) for url in race_card_urls]\n html_contents = await asyncio.gather(*tasks)\n return {\"date\": date,
\"html_contents\": html_contents}\n\n def _parse_races(self, raw_data: Any) -> List[Race]:\n \"\"\"Parses a list of raw HTML
strings into Race objects.\"\"\"\n if not raw_data or not raw_data.get(\"html_contents\"):\n return []\n\n date =
raw_data[\"date\"]\n html_contents = raw_data[\"html_contents\"]\n all_races: List[Race] = []\n\n for html in html_contents:\n
if not html:\n continue\n try:\n parser = HTMLParser(html)\n\n venue_node = (\n
parser.css_first('*[data-test-selector=\"RC-courseHeader__name\"]')\n or
parser.css_first('a[data-test-selector=\"RC-courseHeader__name\"]')\n or
parser.css_first('a[data-test-selector=\"RC-course__name\"]')\n )\n if not venue_node:\n continue\n venue_raw =
node_text(venue_node)\n venue = normalize_venue_name(venue_raw)\n\n race_time_node = (\n
parser.css_first('*[data-test-selector=\"RC-courseHeader__time\"]')\n or
parser.css_first('span[data-test-selector=\"RC-courseHeader__time\"]')\n or
parser.css_first('span[data-test-selector=\"RC-course__time\"]')\n )\n if not race_time_node:\n continue\n race_time_str =
node_text(race_time_node)\n\n # S5 \u2014 extract race type (independent review item)\n race_type = None\n header_text =
node_text(\n parser.css_first('.rp-raceCourse__panel__race__info')\n or parser.css_first('.RC-course__info')\n or
parser.css_first('.RC-courseHeader')\n )\n rt_match =
re.search(r'(Maiden\\s+\\w+|Claiming|Allowance|Graded\\s+Stakes|Stakes)', header_text, re.I)\n if rt_match: race_type =
rt_match.group(1)\n\n is_handicap = None\n if \"HANDICAP\" in header_text.upper():\n is_handicap = True\n\n race_datetime_str
= f\"{date} {race_time_str}\"\n try:\n start_time = datetime.strptime(race_datetime_str, f\"{DATE_FORMAT} %H:%M\")\n except
ValueError:\n # Handle cases where time might have extra text or different format\n time_match =
re.search(r\"(\\d{1,2}:\\d{2})\", race_time_str)\n if time_match:\n start_time = datetime.strptime(f\"{date}
{time_match.group(1)}\", f\"{DATE_FORMAT} %H:%M\")\n else:\n continue\n\n runners = self._parse_runners(parser)\n\n if venue
and runners:\n race_number = self._get_race_number(parser, start_time)\n race = Race(\n id=f\"rp_{venue.lower().replace(' ',
'')}_{date}_{race_number}\",\n venue=venue,\n race_number=race_number,\n start_time=start_time,\n runners=runners,\n
race_type=race_type,\n is_handicap=is_handicap,\n source=self.source_name,\n )\n all_races.append(race)\n except
(AttributeError, ValueError):\n self.logger.error(\"Failed to parse RacingPost race from HTML content.\", exc_info=True)\n
continue\n return all_races\n\n def _get_race_number(self, parser: HTMLParser, start_time: datetime) -> int:\n \"\"\"Derives
```

```
the race number by finding the active time in the nav bar.\"\"\"\n time_str_to_find = start_time.strftime(\"%H:%M\")\n
time_links = parser.css('a[data-test-selector=\"RC-raceTime\"]')\n for i, link in enumerate(time_links):\n if node_text(link)
== time_str_to_find:\n return i + 1\n return 1\n\n def _parse_runners(self, parser: HTMLParser) -> list[Runner]:\n
\"\"\"Parses all runners from a single race card page.\"\"\"\n runners = []\n runner_nodes = (\n
parser.css('div[data-test-selector=\"RC-runnerCard\"]')\n or parser.css('.RC-runnerRow')\n )\n\n # Betting Forecast Fallback\n
forecast_map = {}\n for group in parser.css('*[data-test-selector=\"RC-bettingForecast_group\"]'):\n group_text =
node_text(group)\n # Format: \"2/1 Horse Name\" or similar\n link =
group.css_first('*[data-test-selector=\"RC-bettingForecast_link\"]')\n if link:\n horse_name = clean_text(node_text(link))\n #
Remove horse name from group_text to get odds\n odds_part = group_text.replace(horse_name, \"\").strip().rstrip(\",\")\n if
val := parse_odds_to_decimal(odds_part):\n forecast_map[horse_name.lower()] = val\n\n for node in runner_nodes:\n if runner :=
self._parse_runner(node, forecast_map):\n runners.append(runner)\n return runners\n\n def _parse_runner(self, node: Node,
forecast_map: Optional[Dict[str, float]] = None) -> Optional[Runner]:\n try:\n number_node = (\n
node.css_first('span[data-test-selector=\"RC-cardPage-runnerNumber-no\"]')\n or
node.css_first('span[data-test-selector=\"RC-runnerNumber\"]')\n or node.css_first('.RC-runnerNumber__no')\n )\n name_node =
(\n node.css_first('a[data-test-selector=\"RC-cardPage-runnerName\"]')\n or
node.css_first('a[data-test-selector=\"RC-runnerName\"]')\n or node.css_first('.RC-runnerName')\n )\n odds_node = (\n
node.css_first('span[data-test-selector=\"RC-cardPage-runnerPrice\"]')\n or
node.css_first('a[data-test-selector=\"RC-cardPage-runnerPrice\"]')\n or
node.css_first('span[data-test-selector=\"RC-runnerPrice\"]')\n or node.css_first('.RC-runnerPrice')\n )\n\n if not
name_node:\n return None\n\n name = clean_text(node_text(name_node))\n\n number = 0\n if number_node:\n number_str =
clean_text(node_text(number_node))\n if number_str:\n num_txt = \"\".join(filter(str.isdigit, number_str))\n if num_txt:\n val
= int(num_txt)\n if val <= 100: number = val\n\n odds_str = clean_text(node_text(odds_node)) if odds_node else \"\"\n
scratched = \"NR\" in odds_str.upper() or \"NON-RUNNER\" in node_text(node).upper()\n\n odds = {}\n win_odds = None\n
odds_source = None\n if not scratched:\n win_odds = parse_odds_to_decimal(odds_str)\n if win_odds is not None:\n odds_source =
\"extracted\"\n\n # Betting Forecast Fallback\n if win_odds is None and forecast_map and name.lower() in forecast_map:\n
win_odds = forecast_map[name.lower()]\n odds_source = \"betting_forecast\"\n\n # Advanced heuristic fallback\n if win_odds is
None:\n win_odds = SmartOddsExtractor.extract_from_node(node)\n if win_odds is not None:\n odds_source =
\"smart_extractor\"\n\n if odds_data := create_odds_data(self.source_name, win_odds):\n odds[self.source_name] = odds_data\n\n
return Runner(number=number, name=name, odds=odds, win_odds=win_odds, odds_source=odds_source, scratched=scratched)\n except
Exception:\n self.logger.warning(\"Could not parse RacingPost runner, skipping.\", exc_info=True)\n return None\n",
"name": "RacingPostAdapter"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "class",
"content": "class RacingPostToteAdapter(BrowserHeadersMixin, DebugMixin, BaseAdapterV3):\n \"\"\"\n Adapter for fetching Tote
dividends and results from Racing Post.\n \"\"\"\n ADAPTER_TYPE = \"results\"\n SOURCE_NAME = \"RacingPostTote\"\n BASE_URL =
\"https://www.racingpost.com\"\n\n def __init__(self, config: Optional[Dict[str, Any]] = None):\n
super().__init__(source_name=self.SOURCE_NAME, base_url=self.BASE_URL, config=config)\n\n def _configure_fetch_strategy(self)
-> FetchStrategy:\n return FetchStrategy(\n primary_engine=BrowserEngine.CURL_CFFI,\n enable_js=True,\n
stealth_mode=StealthMode.CAMOUFLAGE,\n timeout=45\n )\n\n def _get_headers(self) -> dict:\n return
self._get_browser_headers(host=\"www.racingpost.com\")\n\n async def _fetch_data(self, date: str) -> Optional[Dict[str,
Any]]:\n dt = parse_date_string(date)\n date_iso = dt.strftime(\"%Y-%m-%d\")\n url = f\"/results/{date_iso}\"\n resp = await
self.make_request(\"GET\", url, headers=self._get_headers())\n if not resp or not resp.text:\n return None\n\n
self._save_debug_snapshot(resp.text, f\"rp_tote_results_{date}\")\n parser = HTMLParser(resp.text)\n\n # Extract links to
individual race results\n links = set()\n selectors = [\n 'a[data-test-selector=\"RC-meetingItem__link_race\"]',\n
'a[href*=\"/results/\"]',\n '.ui-link.rp-raceCourse__panel__race__time',\n 'a.rp-raceCourse__panel__race__time'\n ]\n
target_venues = getattr(self, \"target_venues\", None)\n for s in selectors:\n for a in parser.css(s):\n href =
a.attributes.get(\"href\")\n if href:\n # Filter by venue\n if target_venues:\n match_found = False\n for v in
target_venues:\n if v in href.lower().replace(\"-\", \"\"):\n match_found = True\n break\n if not match_found:\n v_text =
get_canonical_venue(node_text(a))\n if v_text in target_venues:\n match_found = True\n if not match_found:\n continue\n\n #
Broaden regex to match various RP result link patterns\n if re.search(r\"/results/.*?\\d{5,}\", href) or \\\n
re.search(r\"/results/\\d+/\", href) or \\\n re.search(r\"/\\d{4}-\\d{2}-\\d{2}/\", href) or \\\n len(href.split(\"/\")) >=
4:\n links.add(href if href.startswith(\"http\") else f\"{self.BASE_URL}{href}\")\n\n async def fetch_result_page(link):\n r =
await self.make_request(\"GET\", link, headers=self._get_headers())\n return (link, r.text if r else \"\")\n\n tasks =
[fetch_result_page(link) for link in links]\n pages = await asyncio.gather(*tasks)\n return {\"pages\": pages, \"date\":
date}\n\n def _parse_races(self, raw_data: Any) -> List[Race]:\n if not raw_data or not raw_data.get(\"pages\"):\n return
[]\n\n races = []\n date_str = raw_data[\"date\"]\n\n for link, html_content in raw_data[\"pages\"]:\n if not html_content:\n
continue\n try:\n parser = HTMLParser(html_content)\n race = self._parse_result_page(parser, date_str, link)\n if race:\n
races.append(race)\n except Exception as e:\n self.logger.warning(\"Failed to parse RP result page\", link=link,
error=str(e))\n\n return races\n\n def _parse_result_page(self, parser: HTMLParser, date_str: str, url: str) ->
Optional[Race]:\n venue_node = (\n parser.css_first('*[data-test-selector=\"RC-courseHeader__name\"]')\n or
parser.css_first('a[data-test-selector=\"RC-courseHeader__name\"]')\n or
parser.css_first('a[data-test-selector=\"RC-course__name\"]')\n )\n if not venue_node: return None\n venue =
normalize_venue_name(node_text(venue_node))\n\n time_node = (\n
parser.css_first('*[data-test-selector=\"RC-courseHeader__time\"]')\n or
parser.css_first('span[data-test-selector=\"RC-courseHeader__time\"]')\n or
parser.css_first('span[data-test-selector=\"RC-course__time\"]')\n )\n if not time_node: return None\n time_str =
node_text(time_node)\n\n try:\n start_time = datetime.strptime(f\"{date_str} {time_str}\", f\"{DATE_FORMAT}
%H:%M\").replace(tzinfo=EASTERN)\n except Exception:\n return None\n\n # Extract dividends\n dividends = {}\n tote_container =
parser.css_first('div[data-test-selector=\"RC-toteReturns\"]')\n if not tote_container:\n # Try alternate selector\n
tote_container = parser.css_first('.rp-toteReturns')\n\n if tote_container:\n for row in
(tote_container.css('div.rp-toteReturns__row') or tote_container.css('.rp-toteReturns__row')):\n try:\n label_node =
row.css_first('div.rp-toteReturns__label') or row.css_first('.rp-toteReturns__label')\n val_node =
row.css_first('div.rp-toteReturns__value') or row.css_first('.rp-toteReturns__value')\n if label_node and val_node:\n label =
clean_text(node_text(label_node))\n value = clean_text(node_text(val_node))\n if label and value:\n dividends[label] = value\n
except Exception as e:\n self.logger.debug(\"Failed parsing RP tote row\", error=str(e))\n\n\n # Extract runners
(finishers)\n runners = []\n # Try different row selectors for results\n runner_rows = (\n
parser.css('div[data-test-selector=\"RC-resultRunner\"]')\n or parser.css('.RC-runnerRow')\n or
parser.css('.rp-horseTable__mainRow')\n )\n\n for row in runner_rows:\n name_node = (\n
row.css_first('a[data-test-selector=\"RC-resultRunnerName\"]')\n or
row.css_first('*[data-test-selector=\"RC-cardPage-runnerName\"]')\n or row.css_first('.RC-runnerName')\n or
row.css_first('.rp-horseTable__horse__name')\n )\n if not name_node: continue\n name = clean_text(node_text(name_node))\n\n
pos_node = (\n row.css_first('*[data-test-selector=\"RC-cardPage-runnerPosition\"]')\n or
row.css_first('span.rp-resultRunner__position')\n or row.css_first('.rp-horseTable__pos__number')\n )\n pos =
clean_text(node_text(pos_node)) if pos_node else \"?\"\n\n # Try to find saddle number\n number = 0\n num_node = (\n
row.css_first('*[data-test-selector=\"RC-cardPage-runnerNumber-no\"]')\n or row.css_first('.RC-runnerNumber__no')\n or
row.css_first(\".rp-resultRunner__saddleClothNo\")\n or row.css_first(\".rp-horseTable__saddleClothNo\")\n )\n if num_node:\n
try: number = _safe_int(node_text(num_node))\n except Exception: pass\n\n # Extract SP (Starting Price) odds for audit
comparison\n win_odds = None\n odds_source = None\n sp_node = (\n
row.css_first('*[data-test-selector=\"RC-cardPage-runnerPrice\"]')\n or row.css_first('.RC-runnerPrice')\n or
row.css_first('span[data-test-selector=\"RC-resultRunnerSP\"]')\n or row.css_first('.rp-resultRunner__sp')\n or
row.css_first(\".rp-horseTable__horse__sp\")\n )\n if sp_node:\n win_odds =
parse_odds_to_decimal(clean_text(node_text(sp_node)))\n if win_odds is not None:\n odds_source = \"starting_price\"\n\n
odds_data = {}\n if ov := create_odds_data(self.source_name, win_odds):\n odds_data[self.source_name] = ov\n\n
runners.append(Runner(\n name=name,\n number=number,\n win_odds=win_odds,\n odds=odds_data,\n odds_source=odds_source,\n
metadata={\"position\": pos}\n ))\n\n # Derive race number from header or navigation\n race_num = 1\n # Priority 1: Navigation
bar active time (most reliable on RP)\n time_links = parser.css('a[data-test-selector=\"RC-raceTime\"]')\n found_in_nav =
False\n for i, link in enumerate(time_links):\n cls = link.attributes.get(\"class\", \"\")\n if \"active\" in cls or
\"rp-raceTimeCourseName__time\" in cls:\n race_num = i + 1\n found_in_nav = True\n break\n\n if not found_in_nav:\n # Priority
2: Text search for \"Race X\"\n race_num_match = re.search(r'Race\\s+(\\d+)', node_text(parser))\n if race_num_match:\n
race_num = int(race_num_match.group(1))\n\n race = Race(\n id=f\"rp_tote_{get_canonical_venue(venue)}_{date_str.replace('-',
'')}_R{race_num}\",\n venue=venue,\n race_number=race_num,\n start_time=start_time,\n runners=runners,\n
source=self.source_name,\n metadata={\"dividends\": dividends, \"url\": url}\n )\n return race\n",
"name": "RacingPostToteAdapter"
},
{
"type": "miscellaneous",
"content": "\n# ----------------------------------------\n# MASTER ORCHESTRATOR\n#
----------------------------------------\n\n"
},
{
"type": "async_function",
"content": "async def run_discovery(\n target_dates: List[str],\n window_hours: Optional[int] = 8,\n loaded_races:
Optional[List[Race]] = None,\n adapter_names: Optional[List[str]] = None,\n save_path: Optional[str] = None,\n fetch_only:
bool = False,\n live_dashboard: bool = False,\n track_odds: bool = False,\n region: Optional[str] = None,\n config:
Optional[Dict[str, Any]] = None,\n now: Optional[datetime] = None\n):\n logger = structlog.get_logger(\"run_discovery\")\n
logger.info(\"Running Discovery\", dates=target_dates, window_hours=window_hours)\n\n db = FortunaDB()\n await
db.initialize()\n\n try:\n if now is None:\n now = datetime.now(EASTERN)\n cutoff = now + timedelta(hours=window_hours) if
window_hours else None\n all_races_raw = []\n harvest_summary = {}\n\n # Pre-populate harvest_summary based on region/filter
for visibility\n target_region = region or DEFAULT_REGION\n if target_region == \"USA\":\n target_set =
USA_DISCOVERY_ADAPTERS\n elif target_region == \"INT\":\n target_set = INT_DISCOVERY_ADAPTERS\n else:\n target_set =
GLOBAL_DISCOVERY_ADAPTERS\n\n # Determine which adapters should be visible in the harvest summary\n if adapter_names:\n
visible_adapters = [n for n in adapter_names if n in target_set]\n else:\n visible_adapters = list(target_set)\n\n for
adapter_name in visible_adapters:\n harvest_summary[adapter_name] = {\"count\": 0, \"max_odds\": 0.0, \"trust_ratio\":
0.0}\n\n if loaded_races is not None:\n logger.info(\"Using loaded races\", count=len(loaded_races))\n all_races_raw =
loaded_races\n adapters = []\n # Ensure harvest files exist even for loaded runs\n try:\n harvest_file =
get_writable_path(\"discovery_harvest.json\")\n if not harvest_file.exists():\n with open(harvest_file, \"w\") as f:\n
json.dump(harvest_summary, f)\n except Exception: pass\n else:\n # Auto-discover discovery adapter classes\n adapter_classes =
get_discovery_adapter_classes()\n\n if adapter_names:\n adapter_classes = [c for c in adapter_classes if c.__name__ in
adapter_names or getattr(c, \"SOURCE_NAME\", \"\") in adapter_names]\n\n # Load historical performance scores to prioritize
adapters\n adapter_scores = await db.get_adapter_scores(days=30)\n\n # Prioritize adapters by score (descending), with name as
deterministic tiebreaker\n adapter_classes = sorted(\n adapter_classes,\n key=lambda c: (\n -adapter_scores.get(getattr(c,
\"SOURCE_NAME\", c.__name__), 0),\n getattr(c, \"SOURCE_NAME\", c.__name__)\n )\n )\n\n # Get adapter-specific configs from
global config (GPT5 Improvement)\n adapter_configs = config.get(\"adapters\", {}) if config else {}\n\n adapters = []\n for
cls in adapter_classes:\n try:\n name = cls.SOURCE_NAME if hasattr(cls, \"SOURCE_NAME\") else cls.__name__\n specific_config =
adapter_configs.get(name, {}).copy() # Use copy to avoid shared mutation\n # Merge with basic region config\n
specific_config.update({\"region\": region})\n adapters.append(cls(config=specific_config))\n\n # Optimization: Removed
dynamic doubling of mobile adapters to reduce noise/timeouts (Item 7)\n\n except Exception as e:\n logger.error(\"Failed to
initialize adapter\", adapter=cls.__name__, error=str(e))\n\n try:\n async def fetch_one(a, date_str):\n try:\n races = await
a.get_races(date_str)\n return a.source_name, races\n except Exception as e:\n logger.error(\"Error fetching from adapter\",
adapter=a.source_name, date=date_str, error=str(e))\n return a.source_name, []\n\n fetch_tasks = []\n for d in target_dates:\n
for a in adapters:\n fetch_tasks.append(fetch_one(a, d))\n\n results = await asyncio.gather(*fetch_tasks)\n for adapter_name,
r_list in results:\n all_races_raw.extend(r_list)\n\n # Track count and MaxOdds (Proxy for successful odds fetching)\n m_odds
= 0.0\n for r in r_list:\n for run in r.runners:\n if run.win_odds and run.win_odds > m_odds:\n m_odds =
float(run.win_odds)\n\n if adapter_name not in harvest_summary:\n harvest_summary[adapter_name] = {\"count\": 0, \"max_odds\":
0.0}\n\n harvest_summary[adapter_name][\"count\"] += len(r_list)\n if m_odds > harvest_summary[adapter_name][\"max_odds\"]:\n
harvest_summary[adapter_name][\"max_odds\"] = m_odds\n\n # Find the adapter instance to extract its trust_ratio\n
matching_adapter = next((a for a in adapters if a.source_name == adapter_name), None)\n if matching_adapter:\n
harvest_summary[adapter_name][\"trust_ratio\"] = max(\n harvest_summary[adapter_name].get(\"trust_ratio\", 0.0),\n
getattr(matching_adapter, \"trust_ratio\", 0.0)\n )\n\n logger.info(\"Fetched total races\", count=len(all_races_raw))\n
finally:\n # Save discovery harvest summary for GHA reporting and DB persistence\n try:\n harvest_file =
get_writable_path(\"discovery_harvest.json\")\n # Only create if it doesn't exist or we have data\n if harvest_summary or not
harvest_file.exists():\n with open(harvest_file, \"w\") as f:\n json.dump(harvest_summary, f)\n\n if harvest_summary:\n await
db.log_harvest(harvest_summary, region=region)\n except Exception: pass\n\n # Shutdown adapters\n for a in adapters:\n try:
await a.close()\n except Exception: pass\n\n # Apply time window filter if requested to avoid overloading\n # Initial time
window filtering removed to ensure all unique races are tracked for reporting\n\n # Resilience check (FIX_10)\n
adapter_success_counts = {name: data['count'] for name, data in harvest_summary.items() if isinstance(data, dict) and
data.get('count', 0) > 0}\n active_adapters = list(adapter_success_counts.keys())\n total_fetched =
sum(adapter_success_counts.values())\n\n if not all_races_raw:\n logger.error(\"No races fetched from any adapter. Discovery
aborted.\")\n if save_path:\n try:\n target_save = get_writable_path(save_path)\n with open(target_save, \"w\") as f:\n
json.dump([], f)\n logger.info(\"Saved empty race list to file\", path=str(target_save))\n except Exception as e:\n
logger.error(\"Failed to save empty race list\", error=str(e))\n return\n\n if len(active_adapters) == 1 and total_fetched <
20:\n logger.critical(\"DISCOVERY DEGRADED: only one adapter returned data. Results may be unreliable.\",\n
adapter=active_adapters[0], count=total_fetched)\n\n # Deduplicate\n race_map = {}\n for race in all_races_raw:\n
canonical_venue = get_canonical_venue(race.venue)\n # Use Canonical Venue + Race Number + Date + Discipline as stable key\n st
= race.start_time\n if isinstance(st, str):\n try:\n st = from_storage_format(st.replace('Z', '+00:00'))\n except (ValueError,
TypeError):\n pass\n\n date_str = st.strftime('%y%m%d') if hasattr(st, 'strftime') else \"Unknown\"\n # Include discipline in
key to avoid misclassification\n key = f\"{canonical_venue}|{race.race_number}|{date_str}|{race.discipline}\"\n \n if key not
in race_map:\n race_map[key] = race\n else:\n existing = race_map[key]\n # Merge runners/odds\n for nr in race.runners:\n #
Match by number OR name (if numbers are missing)\n er = next((r for r in existing.runners if (r.number != 0 and r.number ==
nr.number) or (r.name.lower() == nr.name.lower())), None)\n if er:\n for source, odds_data in nr.odds.items():\n if source not
in er.odds:\n er.odds[source] = odds_data\n continue\n existing_odds = er.odds[source]\n new_ts = getattr(odds_data,
'last_updated', None)\n old_ts = getattr(existing_odds, 'last_updated', None)\n if new_ts and old_ts and new_ts > old_ts:\n
er.odds[source] = odds_data\n\n if not er.win_odds and nr.win_odds:\n er.win_odds = nr.win_odds\n if not er.number and
nr.number:\n er.number = nr.number\n else:\n existing.runners.append(nr)\n\n # Update source\n sources = set((existing.source
or \"\").split(\", \"))\n sources.add(race.source or \"Unknown\")\n existing.source = \", \".join(sorted(list(filter(None,
sources))))\n\n unique_races = list(race_map.values())\n logger.info(\"Unique races identified\", count=len(unique_races))\n\n
# GPT5 Improvement: Keep all races within window for analysis, not just one per track.\n # Window broadened to 18 hours to
match grid cutoff (News Mode)\n timing_window_races = []\n now = datetime.now(EASTERN)\n for race in unique_races:\n st =
race.start_time\n if isinstance(st, str):\n try:\n st = from_storage_format(st.replace('Z', '+00:00'))\n except (ValueError,
TypeError):\n continue\n if st.tzinfo is None:\n st = st.replace(tzinfo=EASTERN)\n\n # Calculate Minutes to Post\n diff = st -
now\n mtp = diff.total_seconds() / 60\n\n # Timing window limited to 8 hours to ensure yield is audit-able\n if -45 < mtp <=
480: # 8 hours = 480 mins\n timing_window_races.append(race)\n if mtp <= 45:\n logger.info(f\" \ud83d\udcb0 Found Gold
Candidate: {race.venue} R{race.race_number} ({mtp:.1f} MTP)\")\n else:\n logger.debug(f\" \ud83d\udd2d Found Upcoming
Candidate: {race.venue} R{race.race_number} ({mtp:.1f} MTP)\")\n\n golden_zone_races = timing_window_races\n if not
golden_zone_races:\n logger.warning(\"\ud83d\udd2d No races found in the broadened window (-45m to 8h).\")\n\n
logger.info(\"Total unique races available for analysis\", count=len(unique_races))\n\n # Save raw fetched/merged races if
requested (Save EVERYTHING unique)\n if save_path:\n try:\n target_save = get_writable_path(save_path)\n with
open(target_save, \"w\") as f:\n json.dump([r.model_dump(mode='json') for r in unique_races], f, indent=4)\n
logger.info(\"Saved all unique races to file\", path=str(target_save))\n except Exception as e:\n logger.error(\"Failed to
save races\", error=str(e))\n\n if fetch_only:\n logger.info(\"Fetch-only mode active. Skipping analysis and reporting.\")\n
return\n\n\n # Analyze ALL unique races to ensure Grid is populated with Top 5 info (News Mode)\n analyzer =
SimplySuccessAnalyzer(config=config)\n result = analyzer.qualify_races(unique_races, now=now)\n qualified =
result.get(\"races\", [])\n\n # Generate Grid & Goldmine (Grid uses unique_races for the broader context)\n grid =
generate_summary_grid(qualified, all_races=unique_races)\n logger.info(\"Summary Grid Generated\")\n\n # Generate Field Matrix
for all unique races\n field_matrix = generate_field_matrix(unique_races)\n logger.info(\"Field Matrix Generated\")\n\n # Log
Hot Tips & Fetch recent historical results for the report\n tracker = HotTipsTracker(config=config)\n await
tracker.log_tips(qualified)\n\n historical_goldmines = await tracker.db.get_recent_audited_goldmines(limit=15)\n
historical_report = generate_historical_goldmine_report(historical_goldmines)\n\n gm_report =
generate_goldmine_report(qualified, all_races=unique_races)\n if historical_report:\n gm_report += \"\\n\" +
historical_report\n\n # NEW: Dashboard and Live Tracking\n goldmines = [r for r in qualified if get_field(r, 'metadata',
{}).get('is_goldmine')]\n\n # Calculate today's stats for dashboard\n recent_tips = await
tracker.db.get_recent_tips(limit=100)\n today_str = datetime.now(EASTERN).strftime(DATE_FORMAT)\n today_tips = [t for t in
recent_tips if t.get(\"report_date\", \"\").startswith(today_str)]\n\n cashed = sum(1 for t in today_tips if
t.get(\"verdict\") == \"CASHED\")\n total_tips = len(today_tips)\n profit = sum((t.get(\"net_profit\") or 0.0) for t in
today_tips)\n\n stats = {\n \"tips\": total_tips,\n \"cashed\": cashed,\n \"profit\": profit\n }\n\n # Generate friendly HTML
report\n try:\n html_content = await generate_friendly_html_report(qualified, stats)\n html_path =
get_writable_path(\"fortuna_report.html\")\n html_path.write_text(html_content, encoding=\"utf-8\")\n logger.info(\"Friendly
HTML report generated\", path=str(html_path))\n\n # Launch the report if running as a portable app (not in GHA)\n if not
os.getenv(\"GITHUB_ACTIONS\"):\n try:\n # Use absolute path for reliable opening\n abs_path = html_path.absolute()\n if
sys.platform == \"win32\":\n os.startfile(abs_path)\n else:\n webbrowser.open(f\"file://{abs_path}\")\n except Exception as
e:\n logger.warning(\"Failed to automatically launch report\", error=str(e))\n except Exception as e:\n logger.error(\"Failed
to generate HTML report\", error=str(e))\n\n if live_dashboard:\n try:\n from rich.live import Live\n from rich.console import
Console\n # Check if our custom dashboard exists\n try:\n from dashboard import FortunaDashboard\n dash = FortunaDashboard()\n
dash.update(goldmines, stats)\n\n # Start odds tracker if requested\n if track_odds:\n try:\n from odds_tracker import
LiveOddsTracker\n adapter_classes = get_discovery_adapter_classes()\n odds_tracker = LiveOddsTracker(goldmines,
adapter_classes)\n asyncio.create_task(odds_tracker.start_tracking())\n except ImportError:\n logger.warning(\"LiveOddsTracker
not available\")\n\n await dash.run_live()\n except (ImportError, Exception) as e:\n logger.warning(f\"Rich dashboard
component missing or failed: {e}\")\n # Fallback to simple rich display if possible\n console = Console()\n
console.print(\"\\n\" + grid + \"\\n\")\n except ImportError:\n logger.warning(\"Rich library not available, falling back to
static display\")\n print(\"\\n\" + grid + \"\\n\")\n else:\n # Fallback to static print\n try:\n from dashboard import
print_dashboard\n print_dashboard(goldmines, stats)\n except Exception as e:\n # Silently fallback to standard print if
dashboard fails\n pass\n\n print(\"\\n\" + grid + \"\\n\")\n if historical_report:\n print(\"\\n\" + historical_report +
\"\\n\")\n\n # Always save reports to files (GPT5 Improvement: Defensive guards)\n try:\n with
open(get_writable_path(\"summary_grid.txt\"), \"w\", encoding='utf-8') as f: f.write(grid)\n with
open(get_writable_path(\"field_matrix.txt\"), \"w\", encoding='utf-8') as f: f.write(field_matrix)\n with
open(get_writable_path(\"goldmine_report.txt\"), \"w\", encoding='utf-8') as f: f.write(gm_report)\n except Exception as e:\n
logger.error(\"failed_saving_text_reports\", error=str(e))\n\n # Save qualified races to JSON using atomic write (Improvement
1)\n report_data = {\n \"races\": [r.model_dump(mode='json') for r in qualified],\n \"analysis_metadata\":
result.get(\"criteria\", {}),\n \"timestamp\": to_storage_format(datetime.now(EASTERN)),\n }\n qualified_path =
get_writable_path(\"qualified_races.json\")\n temp_path = qualified_path.with_suffix(\".tmp\")\n try:\n with open(temp_path,
\"w\", encoding='utf-8') as f:\n json.dump(report_data, f, indent=4)\n f.flush()\n os.fsync(f.fileno())\n
temp_path.replace(qualified_path)\n\n # Record freshness in GHA output\n is_fresh =
validate_artifact_freshness(str(qualified_path))\n _write_github_output(\"qualified_fresh\", \"1\" if is_fresh else \"0\")\n
_write_github_output(\"qualified_count\", len(qualified))\n except Exception as e:\n
logger.error(\"failed_saving_qualified_races\", error=str(e))\n\n # NEW: Write GHA Job Summary\n if 'GITHUB_STEP_SUMMARY' in
os.environ:\n try:\n predictions_md = format_predictions_section(qualified)\n # We need a db instance for
format_proof_section\n proof_md = await format_proof_section(tracker.db)\n harvest_md = build_harvest_table(harvest_summary,
\"\ud83d\udef0\ufe0f Discovery Harvest Performance\")\n artifacts_md = format_artifact_links()\n
write_job_summary(predictions_md, harvest_md, proof_md, artifacts_md)\n logger.info(\"GHA Job Summary written\")\n except
Exception as e:\n logger.error(\"Failed to write GHA summary\", error=str(e))\n\n finally:\n await
GlobalResourceManager.cleanup()\n",
"name": "run_discovery"
},
{
"type": "async_function",
"content": "async def start_desktop_app():\n \"\"\"Starts a FastAPI server and opens a webview window for the Fortuna
Dashboard.\"\"\"\n try:\n import uvicorn\n from fastapi import FastAPI\n from fastapi.responses import HTMLResponse\n import
webview\n import threading\n import time\n except ImportError as e:\n print(f\"GUI dependencies missing: {e}. Install with
'pip install fastapi uvicorn pywebview'\")\n return\n\n app = FastAPI(title=\"Fortuna Desktop Intelligence\")\n\n
@app.get(\"/\", response_class=HTMLResponse)\n async def get_dashboard():\n # Retrieve latest Goldmines from the database\n db
```

```
= FortunaDB()\n try:\n async with db.get_connection() as conn:\n try:\n async with conn.execute(\n \"SELECT venue,
race_number, selection_number, predicted_2nd_fav_odds, start_time \"\n \"FROM tips ORDER BY id DESC LIMIT 50\"\n ) as
cursor:\n tips = await cursor.fetchall()\n except Exception as e:\n print(f\"DB query failed: {e}\")\n tips = []\n except
Exception as e:\n print(f\"Failed to connect to database: {e}\")\n tips = []\n\n tips_html = \"\".join([\n
f\"<tr><td>{t[4]}</td><td>{t[0]}</td><td>R{t[1]}</td><td>#{t[2]}</td><td>{t[3]}</td></tr>\"\n for t in tips\n ])\n\n return
f\"\"\"\n <html>\n <head>\n <title>Fortuna Intelligence Desktop</title>\n <style>\n body {{ font-family: 'Segoe UI', Tahoma,
Geneva, Verdana, sans-serif; background: #0f172a; color: #f8fafc; padding: 30px; }}\n .container {{ max-width: 1200px; margin:
auto; }}\n h1 {{ color: #fbbf24; border-bottom: 2px solid #fbbf24; padding-bottom: 10px; text-transform: uppercase;
letter-spacing: 2px; }}\n table {{ width: 100%; border-collapse: collapse; margin-top: 20px; background: #1e293b;
border-radius: 8px; overflow: hidden; }}\n th, td {{ padding: 15px; text-align: left; border-bottom: 1px solid #334155; }}\n
th {{ background: #334155; color: #fbbf24; }}\n tr:hover {{ background: #475569; }}\n .footer {{ margin-top: 30px; font-size:
0.8em; color: #94a3b8; text-align: center; }}\n .btn {{ display: inline-block; background: #fbbf24; color: #0f172a; padding:
10px 20px; border-radius: 5px; text-decoration: none; font-weight: bold; margin-bottom: 20px; }}\n </style>\n <script>\n
setTimeout(() => {{ location.reload(); }}, 30000);\n </script>\n </head>\n <body>\n <div class=\"container\">\n <h1>Fortuna
Intelligence Dashboard</h1>\n <p>Monitoring global racing markets for Goldmine opportunities...</p>\n <a href=\"/\"
class=\"btn\">REFRESH NOW</a>\n <table>\n <thead>\n <tr><th>Time
Discovered</th><th>Venue</th><th>Race</th><th>Selection</th><th>Odds</th></tr>\n </thead>\n <tbody>\n {tips_html or \"<tr><td
colspan='5'>No opportunities found yet. Run discovery to populate the database.</td></tr>\"}\n </tbody>\n </table>\n <div
class=\"footer\">Fortuna Intelligence Monolith - Sci-Fi Future Edition - Auto-refreshing every 30s</div>\n </div>\n </body>\n
</html>\n \"\"\"\n\n def run_server():\n uvicorn.run(app, host=\"127.0.0.1\", port=8013, log_level=\"error\")\n\n # Start
FastAPI in a background thread\n server_thread = threading.Thread(target=run_server, daemon=True)\n server_thread.start()\n\n
# Wait a moment for server to initialize\n time.sleep(2.0)\n\n # Create and start the webview window if server is up\n if
server_thread.is_alive():\n print(\"Launching Fortuna Desktop Window...\")\n webview.create_window('Fortuna Intelligence
Desktop', 'http://127.0.0.1:8013', width=1300, height=900)\n webview.start()\n else:\n print(\"\u26a0\ufe0f Error: GUI Server
failed to start.\")\n",
"name": "start_desktop_app"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "async_function",
"content": "async def ensure_browsers(force_install: bool = False):\n \"\"\"Ensure browser dependencies are available for
scraping.\"\"\"\n\n # Skip Playwright in frozen apps if binary doesn't exist - use HTTP-only adapters\n if is_frozen():\n
playwright_path = os.path.expanduser(\"~\\\\AppData\\\\Local\\\\ms-playwright\")\n if not os.path.exists(playwright_path) and
platform.system() == 'Windows':\n structlog.get_logger().info(\"Running as frozen app - Playwright disabled (binary not
found)\")\n return True\n\n try:\n # Check if playwright is installed and has a chromium binary\n from playwright.async_api
import async_playwright\n async with async_playwright() as p:\n try:\n # We try to launch a headless browser to verify
installation\n browser = await p.chromium.launch(headless=True)\n await browser.close()\n return True\n except Exception as
e:\n structlog.get_logger().debug(\"Playwright launch failed during verification\", error=str(e))\n if is_frozen():\n
structlog.get_logger().info(\"Frozen app: Playwright launch failed, using HTTP-only fallbacks\")\n return True\n except
ImportError:\n structlog.get_logger().debug(\"Playwright not imported\")\n if is_frozen(): return True\n\n if is_frozen():\n
return True\n\n # GPT5 Improvement: Instead of auto-installing, warn the user unless opt-in\n # For now, we will assume it's
NOT opt-in and ask for manual installation\n # because auto-pip-installing can be surprising.\n
structlog.get_logger().warning(\"Browser dependencies (Playwright Chromium) missing.\")\n print(\"\\nBrowser dependencies
missing!\")\n print(\"To use browser-based adapters, please run:\")\n print(f\" {sys.executable} -m pip install
playwright==1.49.1\")\n print(f\" {sys.executable} -m playwright install chromium\")\n print(\"Alternatively, run Fortuna
with: --install-browsers\\n\")\n\n # Check if we should auto-install via flag or environment variable\n if force_install or
os.getenv(\"FORTUNA_AUTO_INSTALL_BROWSERS\") == \"1\":\n structlog.get_logger().info(\"Auto-installing browser dependencies as
requested...\")\n try:\n # Remove version pin to avoid conflicts\n subprocess.run([sys.executable, \"-m\", \"pip\",
\"install\", \"playwright\"], check=True, capture_output=True, text=True)\n subprocess.run([sys.executable, \"-m\",
\"playwright\", \"install\", \"chromium\"], check=True, capture_output=True, text=True)\n
structlog.get_logger().info(\"Browser dependencies installed successfully.\")\n return True\n except
subprocess.CalledProcessError as e:\n structlog.get_logger().error(\"Failed to auto-install browsers\", error=str(e))\n return
False\n\n return True # Continue with HTTP-only adapters\n",
"name": "ensure_browsers"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "async_function",
"content": "async def handle_early_exit_args(args: argparse.Namespace, config: Dict[str, Any]) -> bool:\n \"\"\"Handles CLI
arguments that should trigger an immediate exit (GPT5 Improvement).\"\"\"\n if args.quick_help:\n print_quick_help()\n return
True\n if args.status:\n print_status_card(config)\n return True\n if args.show_log:\n await print_recent_logs()\n return
True\n if args.open_dashboard:\n open_report_in_browser()\n return True\n return False\n",
"name": "handle_early_exit_args"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "async_function",
"content": "async def main_all_in_one():\n # Configure logging at the start of main\n structlog.configure(\n
wrapper_class=structlog.make_filtering_bound_logger(logging.INFO)\n )\n # Ensure DB path env is set if passed via argument or
already in environment\n # Actually, we should probably add a --db-path arg here too for parity with analytics\n config =
load_config()\n logger = structlog.get_logger(\"main\")\n parser = argparse.ArgumentParser(description=\"Fortuna All-In-One -
Professional Racing Intelligence\")\n parser.add_argument(\"--date\", type=str, help=\"Target date (YYMMDD)\")\n
parser.add_argument(\"--hours\", type=int, default=8, help=\"Discovery time window in hours (default: 8)\")\n
parser.add_argument(\"--monitor\", action=\"store_true\", help=\"Run in monitor mode\")\n parser.add_argument(\"--once\",
action=\"store_true\", help=\"Run monitor once\")\n parser.add_argument(\"--region\", type=str, choices=[\"USA\", \"INT\",
\"GLOBAL\"], help=\"Filter by region (USA, INT or GLOBAL)\")\n parser.add_argument(\"--quality\", choices=[\"solid\",
```

\"lousy\"], help=\"Filter by adapter quality (Solid Top 3 vs others)\")\n parser.add_argument(\"--include\", type=str, help=\"Comma-separated adapter names to include\")\n parser.add_argument(\"--save\", type=str, help=\"Save races to JSON file\")\n parser.add_argument(\"--load\", type=str, help=\"Load races from JSON file(s), comma-separated\")\n parser.add_argument(\"--fetch-only\", action=\"store_true\", help=\"Only fetch and save data, skip analysis and reporting\")\n parser.add_argument(\"--db-path\", type=str, help=\"Path to tip history database\")\n parser.add_argument(\"--clear-db\", action=\"store_true\", help=\"Clear all tips from the database and exit\")\n parser.add_argument(\"--gui\", action=\"store_true\", help=\"Start the Fortuna Desktop GUI\")\n parser.add_argument(\"--live-dashboard\", action=\"store_true\", help=\"Show live updating terminal dashboard\")\n parser.add_argument(\"--track-odds\", action=\"store_true\", help=\"Monitor live odds and send notifications\")\n parser.add_argument(\"--status\", action=\"store_true\", help=\"Show application status card and latest metrics\")\n parser.add_argument(\"--show-log\", action=\"store_true\", help=\"Print recent fetch/audit highlights\")\n parser.add_argument(\"--quick-help\", action=\"store_true\", help=\"Show friendly onboarding guide\")\n parser.add_argument(\"--open-dashboard\", action=\"store_true\", help=\"Open the HTML intelligence report in browser\")\n parser.add_argument(\"--install-browsers\", action=\"store_true\", help=\"Install required browser dependencies (Playwright Chromium)\")\n args = parser.parse_args()\n\n # Handle early-exit arguments via helper (GPT5 Fix/Improvement)\n if await handle_early_exit_args(args, config):\n return\n\n if args.db_path:\n os.environ[\"FORTUNA_DB_PATH\"] = args.db_path\n\n # Print status card for all normal runs\n print_status_card(config)\n\n if args.install_browsers:\n await ensure_browsers(force_install=True)\n print(\"Installation complete.\")\n return\n\n if args.gui:\n # Start GUI. It runs its own event loop for the webview.\n await ensure_browsers()\n await start_desktop_app()\n return\n\n if args.clear_db:\n db = FortunaDB()\n await db.clear_all_tips()\n await db.close()\n print(\"Database cleared successfully.\")\n return\n\n adapter_filter = [n.strip() for n in args.include.split(\",\")] if args.include else None\n\n # Use default region if not specified\n if not args.region:\n args.region = config.get(\"region\", {}).get(\"default\", DEFAULT_REGION)\n structlog.get_logger().info(\"Using default region\", region=args.region)\n\n # Region-based adapter filtering\n if args.region:\n if args.region == \"USA\":\n target_set = USA_DISCOVERY_ADAPTERS\n elif args.region == \"INT\":\n target_set = INT_DISCOVERY_ADAPTERS\n else:\n target_set = GLOBAL_DISCOVERY_ADAPTERS\n\n if adapter_filter:\n adapter_filter = [n for n in adapter_filter if n in target_set]\n else:\n adapter_filter = list(target_set)\n\n # Quality-based adapter filtering (Council of Superbrains Strategy)\n if args.quality:\n if args.quality == \"solid\":\n if adapter_filter:\n adapter_filter = [n for n in adapter_filter if n in SOLID_DISCOVERY_ADAPTERS]\n else:\n adapter_filter = list(SOLID_DISCOVERY_ADAPTERS)\n else:\n if adapter_filter:\n adapter_filter = [n for n in adapter_filter if n not in SOLID_DISCOVERY_ADAPTERS]\n else:\n # All adapters except solid\n all_names = [getattr(c, \"SOURCE_NAME\", c.__name__) for c in get_discovery_adapter_classes()]\n adapter_filter = [n for n in all_names if n not in SOLID_DISCOVERY_ADAPTERS]\n\n # Special case: TwinSpires needs to know its region internally if it's not filtered out\n # We can pass the region via config if we were creating adapters manually,\n # but here we use names.\n # Actually, I updated TwinSpiresAdapter to check self.config.get(\"region\").\n # I need to ensure the adapter gets this config.\n\n loaded_races = None\n if args.load:\n loaded_races = []\n for path in args.load.split(\",\"):\n path = path.strip()\n if not os.path.exists(path):\n print(f\"Warning: File not found: {path}\")\n logger.warning(\"Race data file not found\", path=path)\n continue\n try:\n with open(path, \"r\") as f:\n data = json.load(f)\n loaded_races.extend([Race.model_validate(r) for r in data])\n except Exception as e:\n print(f\"Error loading {path}: {e}\")\n logger.error(\"Failed to load race data\", path=path, error=str(e), exc_info=True)\n\n if args.date:\n target_dates = [args.date]\n else:\n now = datetime.now(EASTERN)\n future = now + timedelta(hours=args.hours)\n\n target_dates = [now.strftime(DATE_FORMAT)]\n if future.date() > now.date():\n target_dates.append(future.strftime(DATE_FORMAT))\n\n if args.monitor:\n await ensure_browsers()\n monitor = FavoriteToPlaceMonitor(target_dates=target_dates, config=config)\n # Pass region config to monitor\n monitor.config[\"region\"] = args.region\n if args.once:\n await monitor.run_once(loaded_races=loaded_races, adapter_names=adapter_filter)\n if config.get(\"ui\", {}).get(\"auto_open_report\", True) and not os.getenv(\"GITHUB_ACTIONS\"):\n open_report_in_browser()\n else:\n await monitor.run_continuous() # Continuous mode doesn't support load/filter yet for simplicity\n else:\n await ensure_browsers()\n await run_discovery(\n target_dates,\n window_hours=args.hours,\n loaded_races=loaded_races,\n adapter_names=adapter_filter,\n save_path=args.save,\n fetch_only=args.fetch_only,\n live_dashboard=args.live_dashboard,\n track_odds=args.track_odds,\n region=args.region, # Pass region to run_discovery\n config=config\n)\n # Post-run UI enhancements (Council of Superbrains Directive)\n if config.get(\"ui\", {}).get(\"auto_open_report\", True) and not os.getenv(\"GITHUB_ACTIONS\"):\n open_report_in_browser()\n",
"name": "main_all_in_one"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "unknown",
"content": "if __name__ == \"__main__\":\n if os.getenv(\"DEBUG_SNAPSHOTS\"):\n os.makedirs(\"debug_snapshots\", exist_ok=True)\n \n # Windows Event Loop Policy Fix (Project Hardening)\n if sys.platform == 'win32' and not getattr(sys, 'frozen', False):\n try:\n # For non-frozen mode, we prefer Proactor for full feature support\n asyncio.set_event_loop_policy(asyncio.WindowsProactorEventLoopPolicy())\n except AttributeError:\n try:\n asyncio.set_event_loop_policy(asyncio.WindowsSelectorEventLoopPolicy())\n except AttributeError:\n pass\n\n try:\n asyncio.run(main_all_in_one())\n except KeyboardInterrupt:\n pass\n"
}
]
}