```
{
"memo_type": "monolith_structure",
"source_file": "fortuna.py",
"part": 2,
"total_parts": 3,
"blocks": [
{
"type": "miscellaneous",
"content": "\n# ----------------------------------------\n# RacingPostB2BAdapter\n#
----------------------------------------\n"
},
{
"type": "class",
"content": "class RacingPostB2BAdapter(BaseAdapterV3):\n SOURCE_NAME: ClassVar[str] = \"RacingPostB2B\"\n BASE_URL:
ClassVar[str] = \"https://backend-us-racecards.widget.rpb2b.com\"\n\n def __init__(self, config: Optional[Dict[str, Any]] =
None) -> None:\n super().__init__(source_name=self.SOURCE_NAME, base_url=self.BASE_URL, config=config, enable_cache=True,
cache_ttl=300.0, rate_limit=5.0)\n\n def _configure_fetch_strategy(self) -> FetchStrategy:\n return
FetchStrategy(primary_engine=BrowserEngine.HTTPX, enable_js=False, max_retries=3, timeout=20)\n\n async def _fetch_data(self,
date: str) -> Optional[Dict[str, Any]]:\n endpoint = f\"/v2/racecards/daily/{date}\"\n resp = await self.make_request(\"GET\",
endpoint)\n if not resp: return None\n try: data = resp.json()\n except Exception: return None\n if not isinstance(data,
list): return None\n return {\"venues\": data, \"date\": date, \"fetched_at\": datetime.now(EASTERN).isoformat()}\n\n def
_parse_races(self, raw_data: Optional[Dict[str, Any]]) -> List[Race]:\n if not raw_data or not raw_data.get(\"venues\"):
return []\n races: List[Race] = []\n for vd in raw_data[\"venues\"]:\n if vd.get(\"isAbandoned\"): continue\n vn, cc, rd =
vd.get(\"name\", \"Unknown\"), vd.get(\"countryCode\", \"USA\"), vd.get(\"races\", [])\n for r in rd:\n if
r.get(\"raceStatusCode\") == \"ABD\": continue\n parsed = self._parse_single_race(r, vn, cc)\n if parsed:
races.append(parsed)\n return races\n\n def _parse_single_race(self, rd: Dict[str, Any], vn: str, cc: str) ->
Optional[Race]:\n rid, rnum, dts, nr = rd.get(\"id\"), rd.get(\"raceNumber\"), rd.get(\"datetimeUtc\"),
rd.get(\"numberOfRunners\", 0)\n if not all([rid, rnum, dts]): return None\n try: st =
datetime.fromisoformat(dts.replace(\"Z\", \"+00:00\"))\n except Exception: return None\n # Only return race if we have real
runners (avoid placeholder generic runners)\n runners = []\n if runners_raw := rd.get(\"runners\"):\n for i, run_data in
enumerate(runners_raw):\n name = run_data.get(\"name\") or f\"Runner {i+1}\"\n num = run_data.get(\"number\") or i + 1\n
runners.append(Runner(number=num, name=name))\n\n if not runners:\n return None\n\n return Race(discipline=\"Thoroughbred\",
id=f\"rpb2b_{rid.replace('-', '')[:16]}\", venue=normalize_venue_name(vn), race_number=rnum, start_time=st, runners=runners,
source=self.source_name, metadata={\"original_race_id\": rid, \"country_code\": cc, \"num_runners\": nr})\n",
"name": "RacingPostB2BAdapter"
},
{
"type": "miscellaneous",
"content": "\n\n# ----------------------------------------\n# StandardbredCanadaAdapter\n#
----------------------------------------\n"
},
{
"type": "class",
"content": "class StandardbredCanadaAdapter(BrowserHeadersMixin, DebugMixin, RacePageFetcherMixin, BaseAdapterV3):\n
SOURCE_NAME: ClassVar[str] = \"StandardbredCanada\"\n BASE_URL: ClassVar[str] = \"https://standardbredcanada.ca\"\n\n def
__init__(self, config: Optional[Dict[str, Any]] = None) -> None:\n super().__init__(source_name=self.SOURCE_NAME,
base_url=self.BASE_URL, config=config)\n self._semaphore = asyncio.Semaphore(3)\n\n def _configure_fetch_strategy(self) ->
FetchStrategy:\n # Use CURL_CFFI for robust HTTPS and connection handling\n return
FetchStrategy(primary_engine=BrowserEngine.CURL_CFFI, enable_js=False, stealth_mode=\"fast\", timeout=45)\n\n def
_get_headers(self) -> Dict[str, str]:\n return self._get_browser_headers(host=\"standardbredcanada.ca\",
referer=\"https://standardbredcanada.ca/racing\")\n\n async def _fetch_data(self, date: str) -> Optional[Dict[str, Any]]:\n dt
= datetime.strptime(date, \"%Y-%m-%d\")\n date_label = dt.strftime(f\"%A %b {dt.day}, %Y\")\n date_short =
dt.strftime(\"%m%d\") # e.g. 0208\n\n index_html = None\n\n # 1. Try browser-based fetch if available\n try:\n from
playwright.async_api import async_playwright\n async with async_playwright() as p:\n browser = await
p.chromium.launch(headless=True)\n page = await browser.new_page()\n try:\n await page.goto(f\"{self.base_url}/entries\",
wait_until=\"networkidle\")\n await page.evaluate(\"() => { document.querySelectorAll('details').forEach(d => d.open = true);
}\")\n try: await page.select_option(\"#edit-entries-track\", label=\"View All Tracks\")\n except Exception: pass\n try: await
page.select_option(\"#edit-entries-date\", label=date_label)\n except Exception: pass\n try: await
page.click(\"#edit-custom-submit-entries\", force=True, timeout=5000)\n except Exception: pass\n try: await
page.wait_for_selector(\"#entries-results-container a[href*='/entries/']\", timeout=10000)\n except Exception: pass\n
index_html = await page.content()\n finally:\n await page.close()\n await browser.close()\n except Exception as e:\n
self.logger.debug(\"Playwright index fetch failed, trying fallback\", error=str(e))\n\n # 2. Fallback: Try to guess the data
URL pattern if index fetch failed\n if not index_html:\n # Common tracks and their codes (heuristic)\n tracks = [\n (\"Western
Fair\", f\"e{date_short}lonn.dat\"),\n (\"Mohawk\", f\"e{date_short}wbsbsn.dat\"),\n (\"Flamboro\",
f\"e{date_short}flmn.dat\"),\n (\"Rideau\", f\"e{date_short}ridcn.dat\"),\n ]\n metadata = []\n for track_name, filename in
tracks:\n url = f\"/racing/entries/data/{filename}\"\n metadata.append({\"url\": url, \"venue\": track_name, \"finalized\":
True})\n\n pages = await self._fetch_race_pages_concurrent(metadata, self._get_headers())\n return {\"pages\": pages,
\"date\": date}\n\n if not index_html:\n self.logger.warning(\"No index HTML found\", context=\"StandardbredCanada Index
Fetch\")\n return None\n self._save_debug_snapshot(index_html, f\"sc_index_{date}\")\n parser = HTMLParser(index_html)\n
metadata = []\n for container in parser.css(\"#entries-results-container .racing-results-ex-wrap > div\"):\n tnn =
container.css_first(\"h4.track-name\")\n if not tnn: continue\n tn = clean_text(tnn.text()) or \"\"\n isf = \"*\" in tn or
\"*\" in (clean_text(container.text()) or \"\")\n for link in container.css('a[href*=\"/entries/\"]'):\n if u :=
link.attributes.get(\"href\"):\n metadata.append({\"url\": u, \"venue\": tn.replace(\"*\", \"\").strip(), \"finalized\":
isf})\n if not metadata:\n self.logger.warning(\"No metadata found\", context=\"StandardbredCanada Index Parsing\")\n return
None\n pages = await self._fetch_race_pages_concurrent(metadata, self._get_headers(), semaphore_limit=3)\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\"): return []\n try: race_date = datetime.strptime(raw_data.get(\"date\", \"\"), \"%Y-%m-%d\").date()\n
except Exception: race_date = datetime.now(EASTERN).date()\n races: List[Race] = []\n for item in raw_data[\"pages\"]:\n
html_content = item.get(\"html\")\n if not html_content or (\"Final Changes Made\" not in html_content and not
item.get(\"finalized\")): continue\n track_name = normalize_venue_name(item[\"venue\"])\n for pre in
HTMLParser(html_content).css(\"pre\"):\n text = pre.text()\n race_chunks = re.split(r\"(\\d+)\\s+--\\s+\", text)\n for i in
range(1, len(race_chunks), 2):\n try:\n r = self._parse_single_race(race_chunks[i+1], int(race_chunks[i]), race_date,
track_name)\n if r: races.append(r)\n except Exception: continue\n return races\n\n def _parse_single_race(self, content: str,
race_num: int, race_date: date, track_name: str) -> Optional[Race]:\n tm =
re.search(r\"Post\\s+Time:\\s*(\\d{1,2}:\\d{2}\\s*[APM]{2})\", content, re.I)\n st = None\n if tm:\n try: st =
datetime.combine(race_date, datetime.strptime(tm.group(1), \"%I:%M %p\").time())\n except Exception: pass\n if not st: st =
datetime.combine(race_date, datetime.min.time())\n ab = scrape_available_bets(content)\n dist = \"1 Mile\"\n dm =
re.search(r\"(\\d+(?:/\\d+)?\\s+(?:MILE|MILES|KM|F))\", content, re.I)\n if dm: dist = dm.group(1)\n runners = []\n for line
in content.split(\"\\n\"):\n m = re.search(r\"^\\s*(\\d+)\\s+([^(]+)\", line)\n if m:\n num, name = int(m.group(1)),
m.group(2).strip()\n name = re.sub(r\"\\(L\\)$|\\(L\\)\\s+\", \"\", name).strip()\n sc = \"SCR\" in line or \"Scratched\" in
line\n # Try smarter odds extraction from the line\n wo = SmartOddsExtractor.extract_from_text(line)\n if wo is None:\n om =
re.search(r\"(\\d+-\\d+|[0-9.]+)\\s*$\", line)\n if om: wo = parse_odds_to_decimal(om.group(1))\n\n odds_data = {}\n if ov :=
create_odds_data(self.source_name, wo): odds_data[self.source_name] = ov\n runners.append(Runner(number=num, name=name,
scratched=sc, odds=odds_data, win_odds=wo))\n if not runners: return None\n return Race(discipline=\"Harness\",
id=generate_race_id(\"sc\", track_name, st, race_num, \"Harness\"), venue=track_name, race_number=race_num, start_time=st,
runners=runners, distance=dist, source=self.source_name, available_bets=ab)\n",
"name": "StandardbredCanadaAdapter"
},
{
"type": "miscellaneous",
"content": "\n# ----------------------------------------\n# TabAdapter\n# ----------------------------------------\n"
},
{
"type": "class",
"content": "class TabAdapter(BaseAdapterV3):\n SOURCE_NAME: ClassVar[str] = \"TAB\"\n # Note: api.tab.com.au often has DNS
resolution issues in some environments.\n # api.beta.tab.com.au is more reliable.\n BASE_URL: ClassVar[str] =
\"https://api.beta.tab.com.au/v1/tab-info-service/racing\"\n BASE_URL_STABLE: ClassVar[str] =
\"https://api.tab.com.au/v1/tab-info-service/racing\"\n\n def __init__(self, config: Optional[Dict[str, Any]] = None) ->
None:\n super().__init__(source_name=self.SOURCE_NAME, base_url=self.BASE_URL, config=config, rate_limit=2.0)\n\n def
_configure_fetch_strategy(self) -> FetchStrategy:\n # Switch to CURL_CFFI for TAB API to avoid DNS and TLS issues common in
cloud environments\n return FetchStrategy(primary_engine=BrowserEngine.CURL_CFFI, enable_js=False, stealth_mode=\"fast\",
timeout=45)\n\n async def _fetch_data(self, date: str) -> Optional[Dict[str, Any]]:\n url =
f\"{self.base_url}/dates/{date}/meetings\"\n resp = await self.make_request(\"GET\", url, headers={\"Accept\":
\"application/json\", \"User-Agent\": CHROME_USER_AGENT})\n\n if not resp or resp.status != 200:\n self.logger.info(\"Falling
back to STABLE TAB API\")\n url = f\"{self.BASE_URL_STABLE}/dates/{date}/meetings\"\n resp = await self.make_request(\"GET\",
url, headers={\"Accept\": \"application/json\", \"User-Agent\": CHROME_USER_AGENT})\n\n if not resp: return None\n try: data =
resp.json() if hasattr(resp, \"json\") else json.loads(resp.text)\n except Exception: return None\n if not data or
\"meetings\" not in data: return None\n\n # TAB meetings often only have race headers. We need to fetch each meeting's
details\n # to get runners and odds.\n all_meetings = []\n for m in data[\"meetings\"]:\n try:\n vn = m.get(\"meetingName\")\n
mt = m.get(\"meetingType\")\n if vn and mt:\n # Endpoint for meeting details (includes races and runners)\n m_url =
f\"{self.base_url}/dates/{date}/meetings/{mt}/{vn}?jurisdiction=VIC\"\n m_resp = await self.make_request(\"GET\", m_url,
headers={\"Accept\": \"application/json\", \"User-Agent\": CHROME_USER_AGENT})\n if m_resp:\n try:\n m_data = m_resp.json() if
hasattr(m_resp, \"json\") else json.loads(m_resp.text)\n if m_data:\n all_meetings.append(m_data)\n continue\n except
Exception: pass\n # Fallback to the summary data if detail fetch fails\n all_meetings.append(m)\n except Exception:\n
all_meetings.append(m)\n\n return {\"meetings\": all_meetings, \"date\": date}\n\n def _parse_races(self, raw_data: Any) ->
List[Race]:\n if not raw_data or \"meetings\" not in raw_data: return []\n races: List[Race] = []\n for m in
raw_data[\"meetings\"]:\n vn = normalize_venue_name(m.get(\"meetingName\"))\n mt = m.get(\"meetingType\", \"R\")\n disc =
{\"R\": \"Thoroughbred\", \"H\": \"Harness\", \"G\": \"Greyhound\"}.get(mt, \"Thoroughbred\")\n\n for rd in m.get(\"races\",
[]):\n rn = rd.get(\"raceNumber\")\n rst = rd.get(\"raceStartTime\")\n if not rst or not rn: continue\n\n try: st =
datetime.fromisoformat(rst.replace(\"Z\", \"+00:00\"))\n except Exception: continue\n\n runners = []\n # If detail data was
fetched, extract runners\n for runner_data in rd.get(\"runners\", []):\n name = runner_data.get(\"runnerName\", \"Unknown\")\n
num = runner_data.get(\"runnerNumber\")\n\n # Try to get win odds\n win_odds = None\n fixed_odds =
runner_data.get(\"fixedOdds\", {})\n if fixed_odds:\n win_odds = fixed_odds.get(\"returnWin\") or fixed_odds.get(\"win\")\n\n
odds_dict = {}\n if win_odds:\n if ov := create_odds_data(self.source_name, win_odds):\n odds_dict[self.source_name] = ov\n\n
runners.append(Runner(\n name=name,\n number=num,\n win_odds=win_odds,\n odds=odds_dict,\n
scratched=runner_data.get(\"scratched\", False)\n ))\n\n races.append(Race(\n id=generate_race_id(\"tab\", vn, st, rn,
disc),\n venue=vn,\n race_number=rn,\n start_time=st,\n runners=runners,\n discipline=disc,\n source=self.source_name,\n
available_bets=scrape_available_bets(str(rd))\n ))\n return races\n",
"name": "TabAdapter"
},
{
"type": "miscellaneous",
"content": "\n# ----------------------------------------\n# BetfairDataScientistAdapter\n#
----------------------------------------\n"
},
{
"type": "class",
"content": "class BetfairDataScientistAdapter(JSONParsingMixin, BaseAdapterV3):\n ADAPTER_NAME: ClassVar[str] =
\"BetfairDataScientist\"\n\n def __init__(self, model_name: str = \"Ratings\", url: str =
\"https://www.betfair.com.au/hub/ratings/model/horse-racing/\", config: Optional[Dict[str, Any]] = None) -> None:\n
super().__init__(source_name=f\"{self.ADAPTER_NAME}_{model_name}\", base_url=url, config=config)\n self.model_name =
model_name\n\n def _configure_fetch_strategy(self) -> FetchStrategy:\n return
FetchStrategy(primary_engine=BrowserEngine.HTTPX)\n\n async def _fetch_data(self, date: str) -> Optional[StringIO]:\n endpoint
= f\"?date={date}&presenter=RatingsPresenter&csv=true\"\n resp = await self.make_request(\"GET\", endpoint)\n return
StringIO(resp.text) if resp and resp.text else None\n\n def _parse_races(self, raw_data: Optional[StringIO]) -> List[Race]:\n
if not raw_data: return []\n try:\n df = pd.read_csv(raw_data)\n if df.empty: return []\n df =
df.rename(columns={\"meetings.races.bfExchangeMarketId\": \"market_id\", \"meetings.name\": \"meeting_name\",
\"meetings.races.raceNumber\": \"race_number\", \"meetings.races.runners.runnerName\": \"runner_name\",
\"meetings.races.runners.clothNumber\": \"saddle_cloth\", \"meetings.races.runners.ratedPrice\": \"rated_price\"})\n races:
List[Race] = []\n for mid, group in df.groupby(\"market_id\"):\n ri = group.iloc[0]\n runners = []\n for _, row in
group.iterrows():\n rp, od = row.get(\"rated_price\"), {}\n if pd.notna(rp):\n if ov := create_odds_data(self.source_name,
float(rp)): od[self.source_name] = ov\n runners.append(Runner(name=str(row.get(\"runner_name\", \"Unknown\")),
number=int(row.get(\"saddle_cloth\", 0)), odds=od))\n\n vn = normalize_venue_name(str(ri.get(\"meeting_name\", \"\")))\n\n #
Try to find a start time in the CSV\n start_time = datetime.now(EASTERN)\n for col in [\"meetings.races.startTime\",
\"startTime\", \"start_time\", \"time\"]:\n if col in ri and pd.notna(ri[col]):\n try:\n # Assume UTC and convert to Eastern
if it looks like ISO\n st_val = str(ri[col])\n if \"T\" in st_val:\n start_time =
to_eastern(datetime.fromisoformat(st_val.replace(\"Z\", \"+00:00\")))\n break\n except Exception: pass\n\n
races.append(Race(id=str(mid), venue=vn, race_number=int(ri.get(\"race_number\", 0)), start_time=start_time, runners=runners,
source=self.source_name, discipline=\"Thoroughbred\"))\n return races\n except Exception: return []\n",
"name": "BetfairDataScientistAdapter"
```

```
},
{
"type": "miscellaneous",
"content": "\n# ----------------------------------------\n# EquibaseAdapter\n# ----------------------------------------\n"
},
{
"type": "class",
"content": "class EquibaseAdapter(BrowserHeadersMixin, DebugMixin, RacePageFetcherMixin, BaseAdapterV3):\n SOURCE_NAME:
ClassVar[str] = \"Equibase\"\n BASE_URL: ClassVar[str] = \"https://www.equibase.com\"\n\n def __init__(self, config:
Optional[Dict[str, Any]] = None) -> 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 # Equibase uses Instart Logic / Imperva;
PLAYWRIGHT_LEGACY with network_idle is robust\n return FetchStrategy(\n primary_engine=BrowserEngine.PLAYWRIGHT_LEGACY,\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 # Force chrome120 for Equibase as it's the most reliable impersonation for
Imperva/Cloudflare\n kwargs.setdefault(\"impersonate\", \"chrome120\")\n # Let SmartFetcher/curl_cffi handle headers mostly,
but provide minimal essentials if not already set\n h = kwargs.get(\"headers\", {})\n if \"Referer\" not in h: h[\"Referer\"]
= \"https://www.equibase.com/\"\n kwargs[\"headers\"] = h\n return await super().make_request(method, url, **kwargs)\n\n def
_get_headers(self) -> Dict[str, str]:\n return self._get_browser_headers(host=\"www.equibase.com\")\n\n async def
_fetch_data(self, date: str) -> Optional[Dict[str, Any]]:\n dt = datetime.strptime(date, \"%Y-%m-%d\")\n date_str =
dt.strftime(\"%m%d%y\")\n\n # Try different possible index URLs\n index_urls = [\n f\"/static/entry/index.html?SAP=TN\",\n
f\"/static/entry/index.html\",\n f\"/entries/{date}\",\n f\"/entries/index.cfm?date={dt.strftime('%m/%d/%Y')}\",\n ]\n\n resp
= None\n for url in index_urls:\n # Try multiple impersonations to bypass block (Memory Directive Fix)\n for imp in
[\"chrome120\", \"chrome110\", \"safari15_5\"]:\n try:\n resp = await self.make_request(\"GET\", url, impersonate=imp)\n if
resp and resp.status == 200 and resp.text and len(resp.text) > 1000 and \"Pardon Our Interruption\" not in resp.text:\n
self.logger.info(\"Found Equibase index\", url=url, impersonate=imp)\n break\n else:\n text_len = len(resp.text) if resp and
resp.text else 0\n has_pardon = \"Pardon Our Interruption\" in resp.text if resp and resp.text else False\n
self.logger.debug(\"Equibase candidate blocked or invalid\", url=url, impersonate=imp, len=text_len, has_pardon=has_pardon)\n
resp = None\n except Exception as e:\n self.logger.debug(\"Equibase request exception\", url=url, impersonate=imp,
error=str(e))\n resp = None\n if resp: break\n\n if not resp or not resp.text or resp.status != 200:\n if resp:
self.logger.warning(\"Unexpected status\", status=resp.status, url=getattr(resp, 'url', 'Unknown'))\n return None\n\n
self._save_debug_snapshot(resp.text, f\"equibase_index_{date}\")\n parser, links = HTMLParser(resp.text), []\n\n # New: Look
for links in JSON data within scripts (Common on Equibase)\n # Handles escaped slashes and different path separators\n
script_json_matches = re.findall(r'\"URL\":\"([^\"]+)\"', resp.text)\n for url in script_json_matches:\n # Normalizing
backslashes and escaped slashes in found URLs\n url_norm = url.replace(\"\\\\/\", \"/\").replace(\"\\\\\", \"/\")\n # Restrict
lookahead: ensure link is for the targeted date_str\n if \"/static/entry/\" in url_norm and (date_str in url_norm or
\"RaceCardIndex\" in url_norm):\n links.append(url_norm)\n\n for a in parser.css(\"a\"):\n h = a.attributes.get(\"href\") or
\"\"\n c = a.attributes.get(\"class\") or \"\"\n txt = node_text(a).lower()\n # Normalize backslashes (Project fix for
Equibase path separators)\n h_norm = h.replace(\"\\\\\", \"/\")\n\n # Restrict lookahead: ensure link strictly belongs to
targeted date_str (Project Hardening)\n if \"/static/entry/\" in h_norm and (date_str in h_norm or \"RaceCardIndex\" in
h_norm):\n self.logger.debug(\"Equibase link matched\", href=h_norm)\n links.append(h_norm)\n elif \"entry-race-level\" in c
and date_str in h_norm:\n links.append(h_norm)\n elif (\"race-link\" in c or \"track-link\" in c) and date_str in h_norm:\n
links.append(h_norm)\n elif \"entries\" in txt and \"/static/entry/\" in h_norm and date_str in h_norm:\n
links.append(h_norm)\n\n if not links:\n self.logger.warning(\"No links found\", context=\"Equibase Index Parsing\",
date=date)\n return None\n\n # Fetch initial set of pages\n pages = await self._fetch_race_pages_concurrent([{\"url\": l} for
l in set(links)], self._get_headers(), semaphore_limit=5)\n\n all_htmls = []\n extra_links = []\n try:\n target_date =
datetime.strptime(date, \"%Y-%m-%d\").date()\n except Exception:\n target_date = datetime.now(EASTERN).date()\n\n now =
now_eastern()\n for p in pages:\n html_content = p.get(\"html\")\n if not html_content: continue\n\n # If it's an index page
for a track, we need to extract individual race links\n if \"RaceCardIndex\" in p.get(\"url\", \"\"):\n sub_parser =
HTMLParser(html_content)\n # Only take the \"next\" race link for this track (Memory Directive Fix)\n track_races = []\n for a
in sub_parser.css(\"a\"):\n sh = (a.attributes.get(\"href\") or \"\").replace(\"\\\\\", \"/\")\n if \"/static/entry/\" in sh
and date_str in sh and \"RaceCardIndex\" not in sh:\n # Try to find time in text nearby\n time_txt = \"\"\n parent =
a.parent\n if parent:\n time_txt = node_text(parent)\n track_races.append({\"url\": sh, \"time_txt\": time_txt})\n\n next_race
= None\n for r in track_races:\n # Look for 1:00 PM etc\n tm = re.search(r\"(\\d{1,2}:\\d{2}\\s*[APM]{2})\", r[\"time_txt\"],
re.I)\n if tm:\n try:\n rt = datetime.strptime(tm.group(1).upper(), \"%I:%M %p\").replace(\n year=target_date.year,
month=target_date.month, day=target_date.day, tzinfo=EASTERN\n )\n # Skip if in past (Today only)\n if target_date ==
now.date() and rt < now - timedelta(minutes=5):\n continue\n next_race = r\n break\n except Exception: pass\n\n if
next_race:\n extra_links.append(next_race[\"url\"])\n else:\n all_htmls.append(html_content)\n\n if extra_links:\n
self.logger.info(\"Fetching extra race pages from track index\", count=len(extra_links))\n extra_pages = await
self._fetch_race_pages_concurrent([{\"url\": l} for l in set(extra_links)], self._get_headers(), semaphore_limit=5)\n
all_htmls.extend([p.get(\"html\") for p in extra_pages if p and p.get(\"html\")])\n\n return {\"pages\": all_htmls, \"date\":
date}\n\n def _parse_races(self, raw_data: Any) -> List[Race]:\n if not raw_data or not raw_data.get(\"pages\"): return []\n
ds, races = raw_data.get(\"date\", \"\"), []\n for html_content in raw_data[\"pages\"]:\n if not html_content: continue\n
try:\n p = HTMLParser(html_content)\n vn = p.css_first(\"div.track-information strong\")\n rn =
p.css_first(\"div.race-information strong\")\n pt = p.css_first(\"p.post-time span\")\n if not vn or not rn or not pt:
continue\n venue = clean_text(vn.text())\n rnum_txt = rn.text().replace(\"Race\", \"\").strip()\n if not venue or not
rnum_txt.isdigit(): continue\n st = self._parse_post_time(ds, pt.text().strip())\n ab = scrape_available_bets(html_content)\n
runners = [r for node in p.css(\"table.entries-table tbody tr\") if (r := self._parse_runner(node))]\n if not runners:
continue\n races.append(Race(id=f\"eqb_{venue.lower().replace(' ', '')}_{ds}_{rnum_txt}\", venue=venue,
race_number=int(rnum_txt), start_time=st, runners=runners, source=self.source_name, discipline=\"Thoroughbred\",
available_bets=ab))\n except Exception: continue\n return races\n\n def _parse_runner(self, node: Node) -> Optional[Runner]:\n
try:\n cols = node.css(\"td\")\n if len(cols) < 3: return None\n\n # P1: Try to find number in first col\n number = 0\n
num_text = clean_text(cols[0].text())\n if num_text.isdigit():\n number = int(num_text)\n\n # P2: Horse name usually in 3rd
col, but can vary\n name = None\n for idx in [2, 1, 3]:\n if len(cols) > idx:\n n_text = clean_text(cols[idx].text())\n if
n_text and not n_text.isdigit() and len(n_text) > 2:\n name = n_text\n break\n\n if not name: return None\n\n sc =
\"scratched\" in node.attributes.get(\"class\", \"\").lower() or \"SCR\" in (clean_text(node.text()) or \"\")\n\n odds, wo =
{}, None\n if not sc:\n # Odds column can be 9 or 10 (blind indexing fallback)\n for idx in [9, 8, 10]:\n if len(cols) >
idx:\n o_text = clean_text(cols[idx].text())\n if o_text:\n wo = parse_odds_to_decimal(o_text)\n if wo: break\n\n if wo is
None: wo = SmartOddsExtractor.extract_from_node(node)\n if od := create_odds_data(self.source_name, wo):
odds[self.source_name] = od\n\n return Runner(number=number, name=name, odds=odds, win_odds=wo, scratched=sc)\n except
Exception as e:\n self.logger.debug(\"equibase_runner_parse_failed\", error=str(e))\n return None\n\n def
_parse_post_time(self, ds: str, ts: str) -> datetime:\n try:\n parts = ts.replace(\"Post Time:\", \"\").strip().split()\n if
len(parts) >= 2:\n dt = datetime.strptime(f\"{ds} {parts[0]} {parts[1]}\", \"%Y-%m-%d %I:%M %p\")\n return
dt.replace(tzinfo=EASTERN)\n except Exception: pass\n # Fallback to noon UTC for the given date if time parsing fails\n try:\n
dt = datetime.strptime(ds, \"%Y-%m-%d\")\n return dt.replace(hour=12, minute=0, tzinfo=EASTERN)\n except Exception:\n return
datetime.now(EASTERN)\n",
"name": "EquibaseAdapter"
},
{
"type": "miscellaneous",
"content": "\n# ----------------------------------------\n# TwinSpiresAdapter\n# ----------------------------------------\n"
},
{
"type": "class",
"content": "class TwinSpiresAdapter(JSONParsingMixin, DebugMixin, BaseAdapterV3):\n SOURCE_NAME: ClassVar[str] =
\"TwinSpires\"\n BASE_URL: ClassVar[str] = \"https://www.twinspires.com\"\n\n RACE_CONTAINER_SELECTORS: ClassVar[List[str]] =
['div[class*=\"RaceCard\"]', 'div[class*=\"race-card\"]', 'div[data-testid*=\"race\"]', 'div[data-race-id]',
'section[class*=\"race\"]', 'article[class*=\"race\"]', \".race-container\", \"[data-race]\",
'div[class*=\"card\"][class*=\"race\" i]', 'div[class*=\"event\"]']\n TRACK_NAME_SELECTORS: ClassVar[List[str]] =
['[class*=\"track-name\"]', '[class*=\"trackName\"]', '[data-track-name]', 'h2[class*=\"track\"]', 'h3[class*=\"track\"]',
\".track-title\", '[class*=\"venue\"]']\n RACE_NUMBER_SELECTORS: ClassVar[List[str]] = ['[class*=\"race-number\"]',
'[class*=\"raceNumber\"]', '[class*=\"race-num\"]', '[data-race-number]', 'span[class*=\"number\"]']\n POST_TIME_SELECTORS:
ClassVar[List[str]] = [\"time[datetime]\", '[class*=\"post-time\"]', '[class*=\"postTime\"]', '[class*=\"mtp\"]',
\"[data-post-time]\", '[class*=\"race-time\"]']\n RUNNER_ROW_SELECTORS: ClassVar[List[str]] = ['tr[class*=\"runner\"]',
'div[class*=\"runner\"]', 'li[class*=\"runner\"]', \"[data-runner-id]\", 'div[class*=\"horse-row\"]', 'tr[class*=\"horse\"]',
'div[class*=\"entry\"]', \".runner-row\", \".horse-entry\"]\n\n def __init__(self, config: Optional[Dict[str, Any]] = None) ->
None:\n super().__init__(source_name=self.SOURCE_NAME, base_url=self.BASE_URL, config=config, enable_cache=True,
cache_ttl=180.0, rate_limit=1.5)\n\n def _configure_fetch_strategy(self) -> FetchStrategy:\n # TwinSpires is heavily
JS-dependent; Playwright is essential\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 async def make_request(self, method: str, url: str,
**kwargs: Any) -> Any:\n # Force chrome120 for TwinSpires to bypass basic bot checks\n kwargs.setdefault(\"impersonate\",
\"chrome120\")\n # Provide common browser-like headers for TwinSpires\n h = kwargs.get(\"headers\", {})\n if \"Referer\" not
in h: h[\"Referer\"] = \"https://www.google.com/\"\n kwargs[\"headers\"] = h\n return await super().make_request(method, url,
**kwargs)\n\n async def _fetch_data(self, date: str) -> Optional[Dict[str, Any]]:\n ard = []\n last_err = None\n\n # Respect
region from config if provided\n target_region = self.config.get(\"region\") # \"USA\", \"INT\", or None for both\n\n async
def fetch_disc(disc, region=\"USA\"):\n suffix = \"\" if region == \"USA\" else \"?region=INT\"\n # Try date-specific URL
first, fallback to todays-races\n # TwinSpires uses YYYY-MM-DD for races URL\n if date ==
datetime.now(EASTERN).strftime(\"%Y-%m-%d\"):\n url = f\"{self.BASE_URL}/bet/todays-races/{disc}{suffix}\"\n else:\n url =
f\"{self.BASE_URL}/bet/races/{date}/{disc}{suffix}\"\n try:\n resp = await self.make_request(\"GET\", url, network_idle=True,
wait_selector='div[class*=\"race\"], [class*=\"RaceCard\"], [class*=\"track\"]')\n if resp and resp.status == 200:\n
self._save_debug_snapshot(resp.text, f\"ts_{disc}_{region}_{date}\")\n dr = self._extract_races_from_page(resp, date)\n for r
in dr: r[\"assigned_discipline\"] = disc.capitalize()\n return dr\n except Exception as e:\n self.logger.error(\"TwinSpires
fetch failed\", discipline=disc, region=region, error=str(e))\n return []\n\n # Fetch both USA and International for all
disciplines\n tasks = []\n for d in [\"thoroughbred\", \"harness\", \"greyhound\"]:\n if target_region in [None, \"USA\"]:\n
tasks.append(fetch_disc(d, \"USA\"))\n if target_region in [None, \"INT\"]:\n tasks.append(fetch_disc(d, \"INT\"))\n results =
await asyncio.gather(*tasks)\n for r_list in results:\n ard.extend(r_list)\n\n if not ard:\n try:\n resp = await
self.make_request(\"GET\", f\"{self.BASE_URL}/bet/todays-races/time\", network_idle=True)\n if resp and resp.status == 200:
ard = self._extract_races_from_page(resp, date)\n except Exception as e: last_err = last_err or e\n if not ard and last_err:
raise last_err\n return {\"races\": ard, \"date\": date, \"source\": self.source_name} if ard else None\n\n def
_extract_races_from_page(self, resp, date: str) -> List[Dict[str, Any]]:\n if Selector is not None:\n page =
Selector(resp.text)\n else:\n self.logger.warning(\"Scrapling Selector not available, falling back to selectolax\")\n page =
HTMLParser(resp.text)\n\n rd = []\n relems, used = [], None\n for s in self.RACE_CONTAINER_SELECTORS:\n try:\n el =
page.css(s)\n if el:\n relems, used = el, s\n break\n except Exception: continue\n\n if not relems:\n return [{\"html\":
resp.text, \"selector\": page, \"track\": \"Unknown\", \"race_number\": 0, \"date\": date, \"full_page\": True}]\n\n
track_counters = defaultdict(int)\n last_track = \"Unknown\"\n\n for i, relem in enumerate(relems, 1):\n try:\n # Handle both
Scrapling Selector and Selectolax Node\n if hasattr(relem, 'html'):\n html_str = str(relem.html)\n elif hasattr(relem,
'raw_html'):\n html_str = relem.raw_html.decode('utf-8', 'ignore') if isinstance(relem.raw_html, bytes) else
str(relem.raw_html)\n else:\n # Last resort for selectolax: reconstruct HTML or use text\n html_str = str(relem)\n\n # Try to
find track name in the card, but fallback to the last seen track\n # (addressing grouped race cards)\n tn =
self._find_with_selectors(relem, self.TRACK_NAME_SELECTORS)\n if tn:\n last_track = tn.strip()\n\n venue = last_track\n\n
track_counters[venue] += 1\n rnum = track_counters[venue] # Track-specific index as default (Fixes Race 20 issue)\n\n rn_txt =
self._find_with_selectors(relem, self.RACE_NUMBER_SELECTORS)\n if rn_txt:\n digits = \"\".join(filter(str.isdigit, rn_txt))\n
if digits: rnum = int(digits)\n\n rd.append({\n \"html\": html_str,\n \"selector\": relem,\n \"track\": venue,\n
\"race_number\": rnum,\n \"post_time_text\": self._find_with_selectors(relem, self.POST_TIME_SELECTORS),\n \"distance\":
self._find_with_selectors(relem, ['[class*=\"distance\"]', '[class*=\"Distance\"]', '[data-distance]', \".race-distance\"]),\n
\"date\": date,\n \"full_page\": False,\n \"available_bets\": scrape_available_bets(html_str)\n })\n except Exception:
continue\n return rd\n\n def _find_with_selectors(self, el, selectors: List[str]) -> Optional[str]:\n for s in selectors:\n
try:\n f = el.css_first(s)\n if f:\n t = node_text(f)\n if t: return t\n except Exception: continue\n return None\n\n def
_parse_races(self, raw_data: Any) -> List[Race]:\n if not raw_data or \"races\" not in raw_data: return []\n rl, ds, parsed =
raw_data[\"races\"], raw_data.get(\"date\", datetime.now(EASTERN).strftime(\"%Y-%m-%d\")), []\n for rd in rl:\n try:\n r =
self._parse_single_race(rd, ds)\n if r and r.runners: parsed.append(r)\n except Exception: continue\n return parsed\n\n def
_parse_single_race(self, rd: dict, ds: str) -> Optional[Race]:\n page = rd.get(\"selector\")\n hc = rd.get(\"html\", \"\")\n
if not page:\n if not hc: return None\n if Selector is not None:\n page = Selector(hc)\n else:\n page = HTMLParser(hc)\n tn,
rnum = rd.get(\"track\", \"Unknown\"), rd.get(\"race_number\", 1)\n st = self._parse_post_time(rd.get(\"post_time_text\"),
page, ds)\n runners = self._parse_runners(page)\n disc = rd.get(\"assigned_discipline\") or detect_discipline(hc)\n ab =
scrape_available_bets(hc)\n return Race(discipline=disc, id=generate_race_id(\"ts\", tn, st, rnum, disc), venue=tn,
race_number=rnum, start_time=st, runners=runners, distance=rd.get(\"distance\"), source=self.source_name,
available_bets=ab)\n\n def _parse_post_time(self, tt: Optional[str], page, ds: str) -> datetime:\n bd = datetime.strptime(ds,
\"%Y-%m-%d\").date()\n if tt:\n p = self._parse_time_string(tt, bd)\n if p: return p\n for s in self.POST_TIME_SELECTORS:\n
try:\n e = page.css_first(s)\n if e:\n # Scrapling attrib vs Selectolax attributes\n da = getattr(e, 'attrib', getattr(e,
'attributes', {})).get('datetime')\n if da:\n try:\n dt = datetime.fromisoformat(da.replace('Z', '+00:00'))\n # Only trust the
date from HTML if it's within 1 day of what we expected\n if abs((dt.date() - bd).days) <= 1:\n return dt\n else:\n
self.logger.debug(\"Suspicious date in HTML datetime attribute\", html_dt=da, expected_date=bd)\n except Exception: pass\n p =
self._parse_time_string(node_text(e), bd)\n if p: return p\n except Exception: continue\n return datetime.combine(bd,
datetime.now(EASTERN).time()) + timedelta(hours=1)\n\n def _parse_time_string(self, ts: str, bd) -> Optional[datetime]:\n if
not ts: return None\n tc = re.sub(r\"\\s+(EST|EDT|CST|CDT|MST|MDT|PST|PDT|ET|PT|CT|MT)$\", \"\", ts, flags=re.I).strip()\n m =
re.search(r\"(\\d+)\\s*(?:min|mtp)\", tc, re.I)\n if m: return now_eastern() + timedelta(minutes=int(m.group(1)))\n\n for f in
['%I:%M %p', '%I:%M%p', '%H:%M', '%I:%M:%S %p']:\n try:\n t = datetime.strptime(tc, f).time()\n # Heuristic: If time is
between 1:00 and 7:00 and no AM/PM was explicitly in the format\n # (or even if it was, but we are suspicious), for US night
tracks like Turfway,\n # it's likely PM. But %I requires %p. If %H was used and gave < 12, check if it should be PM.\n if f ==
'%H:%M' and 1 <= t.hour <= 7:\n # In US horse racing, 1-7 AM is rare, 1-7 PM is common.\n t = t.replace(hour=t.hour + 12)\n\n
```

```
return datetime.combine(bd, t)\n except Exception: continue\n return None\n\n def _parse_runners(self, page) ->
List[Runner]:\n runners = []\n relems = []\n for s in self.RUNNER_ROW_SELECTORS:\n try:\n el = page.css(s)\n if el: relems =
el; break\n except Exception: continue\n for i, e in enumerate(relems):\n try:\n r = self._parse_single_runner(e, i + 1)\n if
r: runners.append(r)\n except Exception: continue\n return runners\n\n def _parse_single_runner(self, e, dn: int) ->
Optional[Runner]:\n # Scrapling Selector has .html property\n es = str(getattr(e, 'html', e))\n sc = any(s in es.lower() for s
in ['scratched', 'scr', 'scratch'])\n num = None\n for s in ['[class*=\"program\"]', '[class*=\"saddle\"]',
'[class*=\"post\"]', '[class*=\"number\"]', '[data-program-number]', 'td:first-child']:\n try:\n ne = e.css_first(s)\n if
ne:\n nt = node_text(ne)\n dig = \"\".join(filter(str.isdigit, nt))\n if dig:\n val = int(dig)\n if val <= 40:\n num = val\n
break\n except Exception: continue\n name = None\n for s in ['[class*=\"horse-name\"]', '[class*=\"horseName\"]',
'[class*=\"runner-name\"]', 'a[class*=\"name\"]', '[data-horse-name]', 'td:nth-child(2)']:\n try:\n ne = e.css_first(s)\n if
ne:\n nt = node_text(ne)\n if nt and len(nt) > 1: name = re.sub(r\"\\(.*\\)\", \"\", nt).strip(); break\n except Exception:
continue\n if not name: return None\n odds, wo = {}, None\n if not sc:\n for s in ['[class*=\"odds\"]', '[class*=\"ml\"]',
'[class*=\"morning-line\"]', '[data-odds]']:\n try:\n oe = e.css_first(s)\n if oe:\n ot = node_text(oe)\n if ot and ot.upper()
not in ['SCR', 'SCRATCHED', '--', 'N/A']:\n wo = parse_odds_to_decimal(ot)\n if od := create_odds_data(self.source_name, wo):
odds[self.source_name] = od; break\n except Exception: continue\n\n # Advanced heuristic fallback\n if wo is None:\n wo =
SmartOddsExtractor.extract_from_node(e)\n if od := create_odds_data(self.source_name, wo): odds[self.source_name] = od\n\n
return Runner(number=num or dn, name=name, scratched=sc, odds=odds, win_odds=wo)\n\n async def cleanup(self):\n await
self.close()\n self.logger.info(\"TwinSpires adapter cleaned up\")\n",
"name": "TwinSpiresAdapter"
},
{
"type": "miscellaneous",
"content": "\n\n# ----------------------------------------\n# ANALYZER LOGIC\n# ----------------------------------------\n\n"
},
{
"type": "assignment",
"content": "log = structlog.get_logger(__name__)\n"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "function",
"content": "def _get_best_win_odds(runner: Runner) -> Optional[Decimal]:\n \"\"\"Gets the best win odds for a runner,
filtering out invalid or placeholder values.\"\"\"\n if not runner.odds:\n # Fallback to win_odds if available\n if
runner.win_odds and is_valid_odds(runner.win_odds):\n return Decimal(str(runner.win_odds))\n\n valid_odds = []\n for
source_data in runner.odds.values():\n # Handle both dict and primitive formats\n if isinstance(source_data, dict):\n win =
source_data.get('win')\n elif hasattr(source_data, 'win'):\n win = source_data.win\n else:\n win = source_data\n\n if
is_valid_odds(win):\n valid_odds.append(Decimal(str(win)))\n\n if valid_odds:\n return min(valid_odds)\n\n # Final fallback to
win_odds if present\n if runner.win_odds and is_valid_odds(runner.win_odds):\n return Decimal(str(runner.win_odds))\n\n return
None\n",
"name": "_get_best_win_odds"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "class",
"content": "class BaseAnalyzer(ABC):\n \"\"\"The abstract interface for all future analyzer plugins.\"\"\"\n\n def
__init__(self, config: Optional[Dict[str, Any]] = None, **kwargs):\n self.logger =
structlog.get_logger(self.__class__.__name__)\n self.config = config or {}\n\n @abstractmethod\n def qualify_races(self,
races: List[Race]) -> Dict[str, Any]:\n \"\"\"The core method every analyzer must implement.\"\"\"\n pass\n",
"name": "BaseAnalyzer"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "class",
"content": "class TrifectaAnalyzer(BaseAnalyzer):\n \"\"\"Analyzes races and assigns a qualification score based on the
'Trifecta of Factors'.\"\"\"\n\n @property\n def name(self) -> str:\n return \"trifecta_analyzer\"\n\n def __init__(\n self,\n
max_field_size: Optional[int] = None,\n min_favorite_odds: float = 0.01,\n min_second_favorite_odds: float = 0.01,\n
**kwargs\n ):\n super().__init__(**kwargs)\n # Use config value if provided and no explicit override (GPT5 Improvement)\n
self.max_field_size = max_field_size or self.config.get(\"analysis\", {}).get(\"max_field_size\", 11)\n self.min_favorite_odds
= Decimal(str(min_favorite_odds))\n self.min_second_favorite_odds = Decimal(str(min_second_favorite_odds))\n self.notifier =
RaceNotifier()\n\n def is_race_qualified(self, race: Race) -> bool:\n \"\"\"A race is qualified for a trifecta if it has at
least 3 non-scratched runners.\"\"\"\n if not race or not race.runners:\n return False\n\n # Apply global timing cutoff (45m
ago, 120m future)\n now = datetime.now(EASTERN)\n past_cutoff = now - timedelta(minutes=45)\n future_cutoff = now +
timedelta(minutes=120)\n st = race.start_time\n if st.tzinfo is None:\n st = st.replace(tzinfo=EASTERN)\n if st < past_cutoff
or st > future_cutoff:\n return False\n\n active_runners = sum(1 for r in race.runners if not r.scratched)\n return
active_runners >= 3\n\n def qualify_races(self, races: List[Race]) -> Dict[str, Any]:\n \"\"\"Scores all races and returns a
dictionary with criteria and a sorted list.\"\"\"\n qualified_races = []\n TRUSTWORTHY_RATIO_MIN =
self.config.get(\"analysis\", {}).get(\"trustworthy_ratio_min\", 0.7)\n\n for race in races:\n if not
self.is_race_qualified(race):\n continue\n\n active_runners = [r for r in race.runners if not r.scratched]\n total_active =
len(active_runners)\n\n # Trustworthiness Airlock (Success Playbook Item)\n if total_active > 0:\n trustworthy_count = sum(1
for r in active_runners if r.metadata.get(\"odds_source_trustworthy\"))\n if trustworthy_count / total_active <
TRUSTWORTHY_RATIO_MIN:\n log.warning(\"Not enough trustworthy odds for Trifecta; skipping\", venue=race.venue,
race=race.race_number, ratio=round(trustworthy_count/total_active, 2))\n continue\n\n # Uniform Odds Check\n all_odds = []\n
for runner in active_runners:\n odds = _get_best_win_odds(runner)\n if odds: all_odds.append(odds)\n\n if len(all_odds) >= 3
and len(set(all_odds)) == 1:\n log.warning(\"Race contains uniform odds; likely placeholder. Skipping Trifecta.\",
venue=race.venue, race=race.race_number)\n continue\n\n score = self._evaluate_race(race)\n if score > 0:\n
race.qualification_score = score\n qualified_races.append(race)\n\n qualified_races.sort(key=lambda r: r.qualification_score,
reverse=True)\n\n criteria = {\n \"max_field_size\": self.max_field_size,\n \"min_favorite_odds\":
float(self.min_favorite_odds),\n \"min_second_favorite_odds\": float(self.min_second_favorite_odds),\n }\n\n log.info(\n
\"Universal scoring complete\",\n total_races_scored=len(qualified_races),\n criteria=criteria,\n )\n\n for race in
qualified_races:\n if race.qualification_score and race.qualification_score >= 85:\n
self.notifier.notify_qualified_race(race)\n\n return {\"criteria\": criteria, \"races\": qualified_races}\n\n def
_evaluate_race(self, race: Race) -> float:\n \"\"\"Evaluates a single race and returns a qualification score.\"\"\"\n # ---
Constants for Scoring Logic ---\n FAV_ODDS_NORMALIZATION = 10.0\n SEC_FAV_ODDS_NORMALIZATION = 15.0\n FAV_ODDS_WEIGHT = 0.6\n
SEC_FAV_ODDS_WEIGHT = 0.4\n FIELD_SIZE_SCORE_WEIGHT = 0.3\n ODDS_SCORE_WEIGHT = 0.7\n\n active_runners = [r for r in
race.runners if not r.scratched]\n\n runners_with_odds = []\n for runner in active_runners:\n best_odds =
_get_best_win_odds(runner)\n if best_odds is not None:\n runners_with_odds.append((runner, best_odds))\n\n if
len(runners_with_odds) < 2:\n if len(active_runners) >= 2:\n # If we have runners but no odds, use fallbacks\n favorite_odds =
Decimal(str(DEFAULT_ODDS_FALLBACK))\n second_favorite_odds = Decimal(str(DEFAULT_ODDS_FALLBACK))\n else:\n return 0.0\n
else:\n runners_with_odds.sort(key=lambda x: x[1])\n favorite_odds = runners_with_odds[0][1]\n second_favorite_odds =
runners_with_odds[1][1]\n\n # --- Calculate Qualification Score (as inspired by the TypeScript Genesis) ---\n # --- Apply hard
filters before scoring ---\n if (\n len(active_runners) > self.max_field_size\n or favorite_odds < Decimal(\"2.0\")\n or
favorite_odds < self.min_favorite_odds\n or second_favorite_odds < self.min_second_favorite_odds\n ):\n return 0.0\n\n
field_score = (self.max_field_size - len(active_runners)) / self.max_field_size\n\n # Normalize odds scores - cap influence of
extremely high odds\n fav_odds_score = min(float(favorite_odds) / FAV_ODDS_NORMALIZATION, 1.0)\n sec_fav_odds_score =
min(float(second_favorite_odds) / SEC_FAV_ODDS_NORMALIZATION, 1.0)\n\n # Weighted average\n odds_score = (fav_odds_score *
FAV_ODDS_WEIGHT) + (sec_fav_odds_score * SEC_FAV_ODDS_WEIGHT)\n field_score = max(0.0, field_score)\n final_score =
(field_score * FIELD_SIZE_SCORE_WEIGHT) + (odds_score * ODDS_SCORE_WEIGHT)\n # To be safe:\n score = round(final_score * 100,
2)\n race.qualification_score = score\n return score\n",
"name": "TrifectaAnalyzer"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "class",
"content": "class TinyFieldTrifectaAnalyzer(TrifectaAnalyzer):\n \"\"\"A specialized TrifectaAnalyzer that only considers
races with 6 or fewer runners.\"\"\"\n\n def __init__(self, **kwargs):\n # Override the max_field_size to 6 for \"tiny field\"
analysis\n # Set low odds thresholds to \"let them through\" as per user request\n super().__init__(max_field_size=6,
min_favorite_odds=0.01, min_second_favorite_odds=0.01, **kwargs)\n\n @property\n def name(self) -> str:\n return
\"tiny_field_trifecta_analyzer\"\n",
"name": "TinyFieldTrifectaAnalyzer"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "class",
"content": "class SimplySuccessAnalyzer(BaseAnalyzer):\n \"\"\"An analyzer that qualifies every race to show maximum successes
(HTTP 200).\"\"\"\n\n @property\n def name(self) -> str:\n return \"simply_success\"\n\n def qualify_races(self, races:
List[Race]) -> Dict[str, Any]:\n \"\"\"Returns races with a perfect score, applying global timing and chalk filters.\"\"\"\n
qualified = []\n now = datetime.now(EASTERN)\n\n # Success Playbook Hardening (Council of Superbrains)\n TRUSTWORTHY_RATIO_MIN
= self.config.get(\"analysis\", {}).get(\"trustworthy_ratio_min\", 0.7)\n\n for race in races:\n # 1. Timing Filter: Relaxed
for \"News\" mode (GPT5: Caller handles strict timing)\n st = race.start_time\n if st.tzinfo is None:\n st =
st.replace(tzinfo=EASTERN)\n\n # Goldmine Detection: 2nd favorite >= 4.5 decimal\n is_goldmine = False\n is_best_bet = False\n
active_runners = [r for r in race.runners if not r.scratched]\n total_active = len(active_runners)\n\n # Trustworthiness
Airlock (Success Playbook Item)\n if total_active > 0:\n trustworthy_count = sum(1 for r in active_runners if
r.metadata.get(\"odds_source_trustworthy\"))\n if trustworthy_count / total_active < TRUSTWORTHY_RATIO_MIN:\n
self.logger.warning(\"Not enough trustworthy odds; skipping race\", venue=race.venue, race=race.race_number,
ratio=round(trustworthy_count/total_active, 2))\n continue\n\n gap12 = 0.0\n all_odds = []\n\n # 1. Collect and Enrich Odds\n
for runner in active_runners:\n odds = _get_best_win_odds(runner)\n if odds is not None:\n # Propagate fresh odds to runner
object for reporting\n runner.win_odds = float(odds)\n all_odds.append(odds)\n\n # Sort odds ascending\n all_odds.sort()\n\n #
Uniform Odds Check: If all runners have identical odds, it's likely a placeholder card (Memory Directive Fix)\n if
len(all_odds) >= 3 and len(set(all_odds)) == 1:\n self.logger.warning(\"Race contains uniform odds; likely placeholder data.
Skipping.\", venue=race.venue, race=race.race_number, odds=float(all_odds[0]))\n continue\n\n # Stability Check: Ensure we
have at least 2 active runners to compare\n if len(active_runners) < 2:\n log.debug(\"Excluding race with < 2 runners\",
venue=race.venue)\n continue\n\n # 2. Derive Selection (2nd favorite) and Top 5\n # Collect valid runners with their enriched
odds\n valid_r_with_odds = sorted(\n [(r, Decimal(str(r.win_odds))) for r in active_runners if r.win_odds is not None],\n
key=lambda x: x[1]\n )\n race.top_five_numbers = \", \".join([str(r[0].number or '?') for r in valid_r_with_odds[:5]])\n\n if
len(valid_r_with_odds) >= 2:\n sec_fav = valid_r_with_odds[1][0]\n race.metadata['selection_number'] = sec_fav.number\n
race.metadata['selection_name'] = sec_fav.name\n\n # 3. Apply Best Bet Logic\n if len(all_odds) >= 2:\n fav, sec =
all_odds[0], all_odds[1]\n gap12 = round(float(sec - fav), 2)\n\n # Enforce gap requirement\n if gap12 <= 0.25:\n
log.debug(\"Insufficient gap detected (1Gap2 <= 0.25), ineligible for Best Bet treatment\", venue=race.venue,
race=race.race_number, gap=gap12)\n else:\n # Goldmine = 2nd Fav >= 4.5, Field <= 11, Gap > 0.25\n if len(active_runners) <=
11 and sec >= Decimal(\"4.5\"):\n is_goldmine = True\n\n # You Might Like = 2nd Fav >= 3.5, Field <= 11, Gap > 0.25\n if
len(active_runners) <= 11 and sec >= Decimal(\"3.5\"):\n is_best_bet = True\n\n race.metadata['predicted_2nd_fav_odds'] =
float(sec)\n else:\n # Fallback if insufficient odds data\n race.metadata['predicted_2nd_fav_odds'] = None\n\n
race.metadata['is_goldmine'] = is_goldmine\n race.metadata['is_best_bet'] = is_best_bet\n race.metadata['1Gap2'] = gap12\n
race.qualification_score = 100.0\n qualified.append(race)\n\n if not qualified:\n log.warning(\"\ud83d\udd2d SimplySuccess
analyzer pass returned 0 qualified races\", input_count=len(races))\n\n return {\n \"criteria\": {\n \"mode\":
\"simply_success\",\n \"timing_filter\": \"45m_past_to_120m_future\",\n \"chalk_filter\": \"disabled\",\n
\"goldmine_threshold\": 4.5\n },\n \"races\": qualified\n }\n",
"name": "SimplySuccessAnalyzer"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "class",
"content": "class AnalyzerEngine:\n \"\"\"Discovers and manages all available analyzer plugins.\"\"\"\n\n def __init__(self,
config: Optional[Dict[str, Any]] = None):\n self.analyzers: Dict[str, Type[BaseAnalyzer]] = {}\n self.config = config or {}\n
self._discover_analyzers()\n\n def _discover_analyzers(self):\n # In a real plugin system, this would inspect a folder.\n #
For now, we register them manually.\n self.register_analyzer(\"trifecta\", TrifectaAnalyzer)\n
self.register_analyzer(\"tiny_field_trifecta\", TinyFieldTrifectaAnalyzer)\n self.register_analyzer(\"simply_success\",
SimplySuccessAnalyzer)\n log.info(\n \"AnalyzerEngine discovered plugins\",\n
available_analyzers=list(self.analyzers.keys()),\n )\n\n def register_analyzer(self, name: str, analyzer_class:
Type[BaseAnalyzer]):\n self.analyzers[name] = analyzer_class\n\n def get_analyzer(self, name: str, **kwargs) ->
BaseAnalyzer:\n analyzer_class = self.analyzers.get(name)\n if not analyzer_class:\n log.error(\"Requested analyzer not
found\", requested_analyzer=name)\n raise ValueError(f\"Analyzer '{name}' not found.\")\n return
analyzer_class(config=self.config, **kwargs)\n",
"name": "AnalyzerEngine"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "class",
"content": "class AudioAlertSystem:\n \"\"\"Plays sound alerts for important events.\"\"\"\n\n def __init__(self):\n
self.sounds = {\n \"high_value\": Path(__file__).resolve().parent / \"assets\" / \"sounds\" / \"alert_premium.wav\",\n }\n
self.enabled = winsound is not None\n\n def play(self, sound_type: str):\n if not self.enabled:\n return\n\n sound_file =
self.sounds.get(sound_type)\n if sound_file and sound_file.exists():\n try:\n winsound.PlaySound(str(sound_file),
winsound.SND_FILENAME | winsound.SND_ASYNC)\n except Exception as e:\n log.warning(\"Could not play sound\", file=sound_file,
error=e)\n",
"name": "AudioAlertSystem"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "class",
"content": "class RaceNotifier:\n \"\"\"Handles sending native notifications and audio alerts for high-value races.\"\"\"\n\n
def __init__(self):\n self.notifier = DesktopNotifier() if HAS_NOTIFICATIONS else None\n self.audio_system =
AudioAlertSystem()\n self.notified_races = set()\n self.notifications_enabled = self.notifier is not None\n if not
self.notifications_enabled:\n log.debug(\"Native notifications disabled (platform not supported or library missing)\")\n\n def
notify_qualified_race(self, race):\n if race.id in self.notified_races:\n return\n\n # Always log the high-value opportunity
regardless of notification setting\n log.info(\n \"High-value opportunity identified\",\n venue=race.venue,\n
race=race.race_number,\n score=race.qualification_score\n )\n\n if not self.notifications_enabled or self.notifier is None:\n
return\n\n title = \"\ud83d\udc0e High-Value Opportunity!\"\n message = f\"{race.venue} - Race {race.race_number}\\nScore:
{race.qualification_score:.0f}%\\nPost Time: {race.start_time.strftime('%I:%M %p')}\"\n\n try:\n # Use keyword arguments for
better compatibility (AI Review Fix)\n self.notifier.send(\n title=title,\n message=message,\n urgency=\"high\" if
race.qualification_score >= 80 else \"normal\"\n )\n self.notified_races.add(race.id)\n
self.audio_system.play(\"high_value\")\n log.info(\"Notification and audio alert sent for high-value race\",
race_id=race.id)\n except Exception as e:\n log.error(\"Failed to send notification\", error=str(e))\n",
"name": "RaceNotifier"
},
{
"type": "miscellaneous",
"content": "\n\n# ----------------------------------------\n"
},
{
"type": "function",
"content": "def get_track_category(races_at_track: List[Any]) -> str:\n \"\"\"Categorize the track as T (Thoroughbred), H
(Harness), or G (Greyhounds).\"\"\"\n if not races_at_track:\n return 'T'\n\n # Never allow any track with a field size above
7 to be G\n has_large_field = False\n for r in races_at_track:\n runners = get_field(r, 'runners', [])\n active_runners =
len([run for run in runners if not get_field(run, 'scratched', False)])\n if active_runners > 7:\n has_large_field = True\n
break\n\n for race in races_at_track:\n source = get_field(race, 'source', '') or \"\"\n race_id = (get_field(race, 'id', '')
or \"\").lower()\n discipline = get_field(race, 'discipline', '') or \"\"\n\n if discipline == \"Harness\" or '_h' in race_id:
return 'H'\n if (discipline == \"Greyhound\" or '_g' in race_id) and not has_large_field:\n return 'G'\n\n source_lower =
source.lower()\n if (\"greyhound\" in source_lower or source in [\"GBGB\", \"Greyhound\", \"AtTheRacesGreyhound\"]) and not
has_large_field:\n return 'G'\n if source in [\"USTrotting\", \"StandardbredCanada\", \"Harness\"] or any(kw in source_lower
for kw in ['harness', 'standardbred', 'trot', 'pace']):\n return 'H'\n\n # Distance consistency check (Disabled - was
mis-identifying Thoroughbred tracks)\n # dist_counts = defaultdict(int)\n # for r in races_at_track:\n # dist = get_field(r,
'distance')\n # if dist:\n # dist_counts[dist] += 1\n # if dist_counts and max(dist_counts.values()) >= 4:\n # return 'H'\n\n
return 'T'\n",
"name": "get_track_category"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "function",
"content": "def generate_fortuna_fives(races: List[Any], all_races: Optional[List[Any]] = None) -> str:\n \"\"\"Generate the
FORTUNA FIVES appendix.\"\"\"\n lines = [\"\", \"\", \"FORTUNA FIVES\", \"-------------\"]\n fives = []\n for race in
(all_races or races):\n runners = get_field(race, 'runners', [])\n field_size = len([r for r in runners if not get_field(r,
'scratched', False)])\n if field_size == 5:\n fives.append(race)\n\n if not fives:\n lines.append(\"No qualifying races.\")\n
return \"\\n\".join(lines)\n\n track_odds_sums = defaultdict(float)\n track_odds_counts = defaultdict(int)\n stats_races =
all_races if all_races is not None else races\n for race in stats_races:\n v = get_field(race, 'venue')\n track =
normalize_venue_name(v)\n for runner in get_field(race, 'runners', []):\n win_odds = get_field(runner, 'win_odds')\n if not
get_field(runner, 'scratched') and win_odds:\n track_odds_sums[track] += float(win_odds)\n track_odds_counts[track] += 1\n\n
track_avgs = {}\n for track, total in track_odds_sums.items():\n count = track_odds_counts[track]\n if count > 0:\n
track_avgs[track] = str(int(total / count))\n\n track_to_nums = defaultdict(list)\n for r in fives:\n v = get_field(r,
'venue')\n if v:\n track_to_nums[normalize_venue_name(v)].append(get_field(r, 'race_number'))\n\n for track in
sorted(track_to_nums.keys()):\n nums = sorted(list(set(track_to_nums[track])))\n avg_str = f\" [{track_avgs[track]}]\" if
track in track_avgs else \"\"\n lines.append(f\"{track}{avg_str}: {', '.join(map(str, nums))}\")\n\n return
\"\\n\".join(lines)\n",
"name": "generate_fortuna_fives"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "function",
"content": "def generate_field_matrix(races: List[Any]) -> str:\n \"\"\"\n Generates a Markdown table matrix of races by Track
and Field Size.\n Cells contain alphabetic race codes (lowercase=normal, uppercase=goldmine).\n \"\"\"\n if not races:\n
return \"No races available for field matrix.\"\n\n # Group races by Track and Field Size\n matrix = defaultdict(lambda:
defaultdict(list))\n\n for r in races:\n track = normalize_venue_name(get_field(r, 'venue'))\n field_size = len([run for run
in get_field(r, 'runners', []) if not get_field(run, 'scratched', False)])\n\n # Only interested in field sizes 3-11 for this
report\n if 3 <= field_size <= 11:\n is_gold = get_field(r, 'metadata', {}).get('is_goldmine', False)\n race_num =
get_field(r, 'race_number')\n matrix[track][field_size].append((race_num, is_gold))\n\n if not matrix:\n return \"No
qualifying races for field matrix (3-11 runners).\"\n\n # Header: Display sizes 3 to 11\n display_sizes = range(3, 12)\n\n
header = \"| TRACK / FIELD | \" + \" | \".join(map(str, display_sizes)) + \" |\"\n separator = \"| :--- | \" + \" |
\".join([\":---:\"] * len(display_sizes)) + \" |\"\n lines = [header, separator]\n\n for track in sorted(matrix.keys()):\n row
= [track]\n for size in display_sizes:\n race_list = matrix[track].get(size, [])\n if race_list:\n # Standardize formatting of
race codes\n code_parts = format_grid_code(race_list, wrap_width=12)\n row.append(\"<br>\".join(code_parts))\n else:\n
row.append(\" \")\n lines.append(\"| \" + \" | \".join(row) + \" |\")\n\n return \"\\n\".join(lines)\n",
"name": "generate_field_matrix"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "function",
"content": "def generate_goldmines(races: List[Any], all_races: Optional[List[Any]] = None) -> str:\n \"\"\"Generate the
GOLDMINE RACES appendix, filtered to Superfecta races.\"\"\"\n lines = [\"\", \"\", \"GOLDMINE RACES\",
\"--------------\"]\n\n # Pre-calculate track categories\n track_categories = {}\n source_races_for_cat = all_races if
all_races is not None else races\n races_by_track = defaultdict(list)\n for r in source_races_for_cat:\n v = get_field(r,
'venue')\n track = normalize_venue_name(v)\n races_by_track[track].append(r)\n for track, tr_races in
races_by_track.items():\n track_categories[track] = get_track_category(tr_races)\n\n def is_superfecta_effective(r):\n
available_bets = get_field(r, 'available_bets', [])\n metadata_bets = get_field(r, 'metadata', {}).get('available_bets', [])\n
if 'Superfecta' in available_bets or 'Superfecta' in metadata_bets:\n return True\n\n track =
normalize_venue_name(get_field(r, 'venue'))\n cat = track_categories.get(track, 'T')\n runners = get_field(r, 'runners', [])\n
field_size = len([run for run in runners if not get_field(run, 'scratched', False)])\n if cat == 'T' and field_size >= 6:\n
return True\n return False\n\n goldmines = [r for r in races if get_field(r, 'metadata', {}).get('is_goldmine') and
is_superfecta_effective(r)]\n\n if not goldmines:\n lines.append(\"No qualifying races.\")\n return \"\\n\".join(lines)\n\n
track_to_nums = defaultdict(list)\n for r in goldmines:\n v = get_field(r, 'venue')\n if v:\n track =
normalize_venue_name(v)\n track_to_nums[track].append(get_field(r, 'race_number'))\n\n # Sort tracks descending by category (T
> H > G)\n cat_map = {'T': 3, 'H': 2, 'G': 1}\n\n formatted_tracks = []\n for track in track_to_nums.keys():\n cat =
track_categories.get(track, 'T')\n display_name = f\"{cat}~{track}\"\n formatted_tracks.append((cat, track, display_name))\n\n
# Sort: Category Descending, then Track Name Ascending\n formatted_tracks.sort(key=lambda x: (-cat_map.get(x[0], 0),
x[1]))\n\n for cat, track, display_name in formatted_tracks:\n nums = sorted(list(set(track_to_nums[track])))\n
lines.append(f\"{display_name}: {', '.join(map(str, nums))}\")\n return \"\\n\".join(lines)\n",
"name": "generate_goldmines"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "function",
"content": "def generate_goldmine_report(races: List[Any], all_races: Optional[List[Any]] = None) -> str:\n \"\"\"Generate a
detailed report for Goldmine races.\"\"\"\n # 1. Reuse category logic\n track_categories = {}\n source_races_for_cat =
all_races if all_races is not None else races\n races_by_track = defaultdict(list)\n for r in source_races_for_cat:\n v =
get_field(r, 'venue')\n track = normalize_venue_name(v)\n races_by_track[track].append(r)\n for track, tr_races in
races_by_track.items():\n track_categories[track] = get_track_category(tr_races)\n\n def is_superfecta_available(r):\n
available_bets = get_field(r, 'available_bets', [])\n metadata_bets = get_field(r, 'metadata', {}).get('available_bets', [])\n
if 'Superfecta' in available_bets or 'Superfecta' in metadata_bets:\n return True\n track = normalize_venue_name(get_field(r,
'venue'))\n cat = track_categories.get(track, 'T')\n runners = get_field(r, 'runners', [])\n field_size = len([run for run in
runners if not get_field(run, 'scratched', False)])\n return cat == 'T' and field_size >= 6\n\n # Include all goldmines (2nd
fav >= 4.5)\n # Deduplicate to prevent double-reporting (e.g. from multiple sources)\n goldmines = []\n seen_gold = set()\n
for r in races:\n if get_field(r, 'metadata', {}).get('is_goldmine'):\n track = get_canonical_venue(get_field(r, 'venue'))\n
num = get_field(r, 'race_number')\n st = get_field(r, 'start_time')\n st_str = st.strftime('%Y%m%d') if isinstance(st,
datetime) else str(st)\n # Use canonical key for cross-adapter deduplication\n key = (track, num, st_str)\n if key not in
seen_gold:\n seen_gold.add(key)\n goldmines.append(r)\n\n if not goldmines:\n return \"No Goldmine races found.\"\n\n # Sort
goldmines: Cat descending, Track asc, Race num asc\n cat_map = {'T': 3, 'H': 2, 'G': 1}\n def goldmine_sort_key(r):\n track =
normalize_venue_name(get_field(r, 'venue'))\n cat = track_categories.get(track, 'T')\n return (-cat_map.get(cat, 0), track,
get_field(r, 'race_number', 0))\n\n goldmines.sort(key=goldmine_sort_key)\n\n now = datetime.now(EASTERN)\n
immediate_gold_superfecta = []\n immediate_gold = []\n remaining_gold = []\n\n for r in goldmines:\n start_time = get_field(r,
'start_time')\n if isinstance(start_time, str):\n try:\n start_time = datetime.fromisoformat(start_time.replace('Z',
'+00:00'))\n except ValueError:\n remaining_gold.append(r)\n continue\n\n if start_time:\n if start_time.tzinfo is None:\n
start_time = start_time.replace(tzinfo=EASTERN)\n\n diff = (start_time - now).total_seconds() / 60\n if 0 <= diff <= 20:\n if
is_superfecta_available(r):\n immediate_gold_superfecta.append(r)\n else:\n immediate_gold.append(r)\n else:\n
remaining_gold.append(r)\n else:\n remaining_gold.append(r)\n\n report_lines = [\"LIST OF BEST BETS - GOLDMINE REPORT\",
\"===================================\", \"\"]\n\n def render_races(races_to_render, label):\n if not races_to_render:\n
return\n report_lines.append(f\"--- {label.upper()} ---\")\n report_lines.append(\"-\" * (len(label) + 8))\n
```

```
report_lines.append(\"\")\n\n for r in races_to_render:\n track = normalize_venue_name(get_field(r, 'venue'))\n cat =
track_categories.get(track, 'T')\n race_num = get_field(r, 'race_number')\n start_time = get_field(r, 'start_time')\n if
isinstance(start_time, datetime):\n # Ensure it's in Eastern for the display\n st_eastern = to_eastern(start_time)\n time_str
= st_eastern.strftime(\"%H:%M ET\")\n else:\n time_str = str(start_time)\n\n # Identify Top 5\n runners = get_field(r,
'runners', [])\n active_with_odds = []\n for run in runners:\n if get_field(run, 'scratched'): continue\n wo =
_get_best_win_odds(run)\n if wo: active_with_odds.append((run, wo))\n\n sorted_by_odds = sorted(active_with_odds, key=lambda
x: x[1])\n top_5_nums = \", \".join([str(get_field(run[0], 'number') or '?') for run in sorted_by_odds[:5]])\n if hasattr(r,
'top_five_numbers'):\n r.top_five_numbers = top_5_nums\n\n gap12 = get_field(r, 'metadata', {}).get('1Gap2', 0.0)\n
report_lines.append(f\"{cat}~{track} - Race {race_num} ({time_str})\")\n report_lines.append(f\"PREDICTED TOP 5:
[{top_5_nums}] | 1Gap2: {gap12:.2f}\")\n report_lines.append(\"-\" * 40)\n\n # Sort runners by number\n sorted_runners =
sorted(runners, key=lambda x: get_field(x, 'number') or 0)\n\n for run in sorted_runners:\n if get_field(run, 'scratched'):\n
continue\n name = get_field(run, 'name')\n num = get_field(run, 'number')\n odds = get_field(run, 'win_odds')\n odds_str =
f\"{odds:.2f}\" if odds else \"N/A\"\n report_lines.append(f\" #{num:<2} {name:<25} ~ {odds_str}\")\n\n
report_lines.append(\"\")\n\n if immediate_gold_superfecta:\n render_races(immediate_gold_superfecta, \"Immediate Gold
(superfecta)\")\n\n if immediate_gold:\n render_races(immediate_gold, \"Immediate Gold\")\n\n if remaining_gold:\n
render_races(remaining_gold, \"All Remaining Goldmine Races\")\n\n return \"\\n\".join(report_lines)\n",
"name": "generate_goldmine_report"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "function",
"content": "def generate_historical_goldmine_report(audited_tips: List[Dict[str, Any]]) -> str:\n \"\"\"Generate a report for
recently audited Goldmine races.\"\"\"\n if not audited_tips:\n return \"\"\n\n lines = [\"\", \"RECENT AUDITED GOLDMINES\",
\"------------------------\"]\n\n # Calculate simple stats\n total = len(audited_tips)\n cashed = sum(1 for t in audited_tips
if t.get(\"verdict\") == \"CASHED\")\n total_profit = sum((t.get(\"net_profit\") or 0.0) for t in audited_tips)\n sr = (cashed
/ total * 100) if total > 0 else 0\n\n lines.append(f\"Performance Summary (Last {total} Goldmines):\")\n lines.append(f\"
Strike Rate: {sr:.1f}% | Total Net Profit: ${total_profit:+.2f}\")\n lines.append(\"\")\n\n for tip in audited_tips:\n venue =
tip.get(\"venue\", \"Unknown\")\n race_num = tip.get(\"race_number\", \"?\")\n verdict = tip.get(\"verdict\", \"?\")\n profit
= tip.get(\"net_profit\", 0.0)\n start_time_raw = tip.get(\"start_time\", \"\")\n\n try:\n st =
datetime.fromisoformat(start_time_raw.replace('Z', '+00:00'))\n time_str = to_eastern(st).strftime(\"%Y-%m-%d %H:%M ET\")\n
except Exception:\n time_str = str(start_time_raw)[:16]\n\n emoji = \"\u2705\" if verdict == \"CASHED\" else \"\u274c\" if
verdict == \"BURNED\" else \"\u26aa\"\n\n line = f\"{emoji} {time_str} | {venue} R{race_num} | {verdict:<6} | Profit:
${profit:+.2f}\"\n\n # Add top place payouts for proof\n p1 = tip.get(\"top1_place_payout\")\n p2 =
tip.get(\"top2_place_payout\")\n if p1 or p2:\n line += f\" | Place: {p1 or 0:.2f}/{p2 or 0:.2f}\"\n\n # Prioritize Superfecta
info to \"prove\" with payouts\n super_payout = tip.get(\"superfecta_payout\")\n tri_payout = tip.get(\"trifecta_payout\")\n\n
if super_payout:\n line += f\" | Super: ${super_payout:.2f}\"\n elif tri_payout:\n line += f\" | Tri: ${tri_payout:.2f}\"\n\n
lines.append(line)\n\n return \"\\n\".join(lines)\n",
"name": "generate_historical_goldmine_report"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "function",
"content": "def generate_next_to_jump(races: List[Any]) -> str:\n \"\"\"Generate the NEXT TO JUMP section.\"\"\"\n lines =
[\"\", \"\", \"NEXT TO JUMP\", \"------------\"]\n now = datetime.now(EASTERN)\n upcoming = []\n for r in races:\n r_time =
get_field(r, 'start_time')\n if isinstance(r_time, str):\n try:\n r_time = datetime.fromisoformat(r_time.replace('Z',
'+00:00'))\n except ValueError:\n continue\n\n if r_time:\n if r_time.tzinfo is None:\n r_time =
r_time.replace(tzinfo=EASTERN)\n if r_time > now:\n upcoming.append((r, r_time))\n\n if upcoming:\n next_r, next_r_time =
min(upcoming, key=lambda x: x[1])\n diff = next_r_time - now\n minutes = int(diff.total_seconds() / 60)\n
lines.append(f\"{normalize_venue_name(get_field(next_r, 'venue'))} Race {get_field(next_r, 'race_number')} in {minutes}m\")\n
else:\n lines.append(\"All races complete for today.\")\n\n return \"\\n\".join(lines)\n",
"name": "generate_next_to_jump"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "async_function",
"content": "async def generate_friendly_html_report(races: List[Any], stats: Dict[str, Any]) -> str:\n \"\"\"Generates a
high-impact, friendly HTML report for the Fortuna Faucet.\"\"\"\n now_str = datetime.now(EASTERN).strftime('%Y-%m-%d
%H:%M:%S')\n\n # 1. Best Bet Opportunities\n rows = []\n for r in sorted(races, key=lambda x: getattr(x, 'start_time', '')):\n
# Get selection (2nd favorite)\n runners = getattr(r, 'runners', [])\n active = [run for run in runners if not getattr(run,
'scratched', False)]\n if len(active) < 2: continue\n\n active.sort(key=lambda x: getattr(x, 'win_odds', 999.0) or 999.0)\n
sel = active[1]\n\n st = getattr(r, 'start_time', '')\n if isinstance(st, datetime):\n # Ensure it's in Eastern for display
(GPT5 Improvement)\n st_str = to_eastern(st).strftime('%H:%M')\n elif isinstance(st, str):\n try:\n dt =
datetime.fromisoformat(st.replace('Z', '+00:00'))\n st_str = to_eastern(dt).strftime('%H:%M')\n except Exception:\n st_str =
str(st)[11:16]\n else:\n st_str = str(st)[11:16]\n\n is_gold = getattr(r, 'metadata', {}).get('is_goldmine', False)\n
gold_badge = '<span class=\"badge gold\">GOLD</span>' if is_gold else ''\n\n d_str = '??/??'\n if isinstance(st, datetime):\n
d_str = st.strftime('%m/%d')\n elif isinstance(st, str):\n try:\n dt = datetime.fromisoformat(st.replace('Z', '+00:00'))\n
d_str = dt.strftime('%m/%d')\n except Exception: pass\n\n rows.append(f\"\"\"\n <tr>\n <td>{st_str} ({d_str})</td>\n
<td>{getattr(r, 'venue', 'Unknown')}</td>\n <td>R{getattr(r, 'race_number', '?')}</td>\n <td>#{getattr(sel, 'number', '?')}
{getattr(sel, 'name', 'Unknown')}</td>\n <td>{ (getattr(sel, 'win_odds') or 0.0):.2f}</td>\n <td>{gold_badge}</td>\n </tr>\n
\"\"\")\n\n tips_count = stats.get('tips', 0)\n cashed_count = stats.get('cashed', 0)\n profit = stats.get('profit', 0.0)\n\n
html = f\"\"\"\n <!DOCTYPE html>\n <html lang=\"en\">\n <head>\n <meta charset=\"UTF-8\">\n <meta name=\"viewport\"
content=\"width=device-width, initial-scale=1.0\">\n <title>Fortuna Faucet Intelligence Report</title>\n <style>\n body {{
font-family: 'Segoe UI', Arial, sans-serif; background-color: #0f172a; color: #f8fafc; margin: 0; padding: 20px; }}\n
.container {{ max-width: 1000px; margin: 0 auto; background-color: #1e293b; padding: 30px; border-radius: 12px; box-shadow: 0
10px 25px rgba(0,0,0,0.5); }}\n h1 {{ color: #fbbf24; text-align: center; text-transform: uppercase; letter-spacing: 3px;
border-bottom: 2px solid #fbbf24; padding-bottom: 15px; }}\n .stats-grid {{ display: grid; grid-template-columns: repeat(3,
1fr); gap: 20px; margin: 30px 0; }}\n .stat-card {{ background-color: #334155; padding: 20px; border-radius: 8px; text-align:
center; }}\n .stat-value {{ font-size: 24px; font-weight: bold; color: #fbbf24; }}\n .stat-label {{ font-size: 14px; color:
#94a3b8; text-transform: uppercase; }}\n table {{ width: 100%; border-collapse: collapse; margin-top: 20px; }}\n th {{
background-color: #334155; color: #fbbf24; text-align: left; padding: 12px; }}\n td {{ padding: 12px; border-bottom: 1px solid
#334155; }}\n tr:hover {{ background-color: #334155; }}\n .badge {{ padding: 4px 8px; border-radius: 4px; font-size: 12px;
font-weight: bold; }}\n .gold {{ background-color: #fbbf24; color: #0f172a; }}\n .footer {{ margin-top: 40px; text-align:
center; font-size: 12px; color: #64748b; }}\n </style>\n </head>\n <body>\n <div class=\"container\">\n <h1>Fortuna Faucet
Intelligence</h1>\n <p style=\"text-align:center;\">Real-time global racing analysis generated at {now_str} ET</p>\n\n <div
class=\"stats-grid\">\n <div class=\"stat-card\">\n <div class=\"stat-value\">{tips_count}</div>\n <div
class=\"stat-label\">Total Selections</div>\n </div>\n <div class=\"stat-card\">\n <div
class=\"stat-value\">{cashed_count}</div>\n <div class=\"stat-label\">Recently Audited Wins</div>\n </div>\n <div
class=\"stat-card\">\n <div class=\"stat-value\">${profit:+.2f}</div>\n <div class=\"stat-label\">Estimated Profit</div>\n
</div>\n </div>\n\n <h2>\ud83d\udd25 Best Bet Opportunities</h2>\n <table>\n <thead>\n <tr>\n <th>Time</th>\n <th>Venue</th>\n
<th>Race</th>\n <th>Selection</th>\n <th>Odds</th>\n <th>Type</th>\n </tr>\n </thead>\n <tbody>\n {''.join(rows) if rows else
'<tr><td colspan=\"6\" style=\"text-align:center;\">No immediate opportunities identified.</td></tr>'}\n </tbody>\n
</table>\n\n {await _generate_audit_history_html()}\n\n <div class=\"footer\">\n Fortuna Faucet Portable App - Sci-Fi
Intelligence Edition<br>\n Powered by the Council of Superbrains\n </div>\n </div>\n </body>\n </html>\n \"\"\"\n return
html\n",
"name": "generate_friendly_html_report"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "async_function",
"content": "async def _generate_audit_history_html() -> str:\n \"\"\"Generates HTML for recent audited results.\"\"\"\n db =
FortunaDB()\n history = await db.get_all_audited_tips()\n if not history:\n return \"\"\n\n # Take latest 15\n history =
sorted(history, key=lambda x: x.get('audit_timestamp', ''), reverse=True)[:15]\n\n rows = []\n for t in history:\n verdict =
t.get(\"verdict\", \"?\")\n emoji = \"\u2705\" if verdict == \"CASHED\" else \"\u274c\" if verdict == \"BURNED\" else
\"\u26aa\"\n profit = t.get(\"net_profit\", 0.0)\n p_class = \"profit-pos\" if profit > 0 else \"profit-neg\" if profit < 0
else \"\"\n\n po = t.get(\"predicted_2nd_fav_odds\")\n ao = t.get(\"actual_2nd_fav_odds\")\n odds_str = f\"{po or '?':.1f}
\u2192 {ao or '?':.1f}\"\n\n rows.append(f\"\"\"\n <tr>\n <td>{emoji} {verdict}</td>\n <td>{t.get('venue', 'Unknown')}</td>\n
<td>R{t.get('race_number', '?')}</td>\n <td>{odds_str}</td>\n <td class=\"{p_class}\">${profit:+.2f}</td>\n </tr>\n
\"\"\")\n\n return f\"\"\"\n <style>\n .profit-pos {{ color: #4ade80; font-weight: bold; }}\n .profit-neg {{ color: #f87171;
}}\n </style>\n <h2 style=\"margin-top: 40px;\">\ud83d\udcb0 Recent Audit Results</h2>\n <table>\n <thead>\n <tr>\n
<th>Verdict</th>\n <th>Venue</th>\n <th>Race</th>\n <th>Odds (Pred \u2192 Act)</th>\n <th>Net Profit</th>\n </tr>\n </thead>\n
<tbody>\n {''.join(rows)}\n </tbody>\n </table>\n \"\"\"\n",
"name": "_generate_audit_history_html"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "function",
"content": "def generate_summary_grid(races: List[Any], all_races: Optional[List[Any]] = None) -> str:\n \"\"\"\n Generates a
Markdown table summary of upcoming races.\n Sorted by MTP, ceiling of 4 hours from now.\n \"\"\"\n now =
datetime.now(EASTERN)\n cutoff = now + timedelta(hours=18)\n\n # 1. Pre-calculate track categories\n track_categories = {}\n
source_races = all_races if all_races is not None else races\n races_by_track = defaultdict(list)\n for r in source_races:\n
venue = get_field(r, 'venue')\n track = normalize_venue_name(venue)\n races_by_track[track].append(r)\n\n for track, tr_races
in races_by_track.items():\n track_categories[track] = get_track_category(tr_races)\n\n table_races = []\n seen = set()\n for
race in (all_races or races):\n st = get_field(race, 'start_time')\n if isinstance(st, str):\n try: st =
datetime.fromisoformat(st.replace('Z', '+00:00'))\n except Exception: continue\n if st and st.tzinfo is None: st =
st.replace(tzinfo=EASTERN)\n\n # Ceiling of 18 hours, ignore races more than 10 mins past\n if not st or st < now -
timedelta(minutes=10) or st > cutoff:\n continue\n\n track = normalize_venue_name(get_field(race, 'venue'))\n canonical_track
= get_canonical_venue(get_field(race, 'venue'))\n num = get_field(race, 'race_number')\n # Deduplication key: Use canonical
track/num/date\n key = (canonical_track, num, st.strftime('%Y%m%d'))\n if key in seen: continue\n seen.add(key)\n\n mtp =
int((st - now).total_seconds() / 60)\n runners = get_field(race, 'runners', [])\n field_size = len([run for run in runners if
not get_field(run, 'scratched', False)])\n top5 = getattr(race, 'top_five_numbers', 'N/A')\n gap12 = get_field(race,
'metadata', {}).get('1Gap2', 0.0)\n is_gold = get_field(race, 'metadata', {}).get('is_goldmine', False)\n\n
table_races.append({\n 'mtp': mtp,\n 'cat': track_categories.get(track, 'T'),\n 'track': track,\n 'num': num,\n 'field':
field_size,\n 'top5': top5,\n 'gap': gap12,\n 'gold': '[G]' if is_gold else ''\n })\n\n # Sort by MTP\n
table_races.sort(key=lambda x: x['mtp'])\n\n if not table_races:\n return \"No upcoming races in the next 4 hours.\"\n\n lines
= [\n \"| MTP | CAT | TRACK | R# | FLD | TOP 5 | GAP | |\",\n \"|:---:|:---:|:---|:---:|:---:|:---|:---:|:---:|\"\n ]\n for tr
in table_races:\n # Better alignment: leading zero for single digits (Memory Directive Fix)\n mtp_val = tr['mtp']\n mtp_str =
f\"{mtp_val:02d}\" if 0 <= mtp_val < 10 else str(mtp_val)\n lines.append(f\"| {mtp_str}m | {tr['cat']} | {tr['track'][:20]} |
{tr['num']} | {tr['field']} | `{tr['top5']}` | {tr['gap']:.2f} | {tr['gold']} |\")\n\n return \"\\n\".join(lines)\n",
"name": "generate_summary_grid"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "function",
"content": "def normalize_course_name(name: str) -> str:\n if not name:\n return \"\"\n name = name.lower().strip()\n name =
re.sub(r\"[^a-z0-9\\s-]\", \"\", name)\n name = re.sub(r\"[\\s-]+\", \"_\", name)\n return name\n",
"name": "normalize_course_name"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
```

```
"type": "function",
"content": "def num_to_alpha(n, is_goldmine=False):\n \"\"\"Convert race number to alphabetic code. Goldmines are
uppercase.\"\"\"\n if not isinstance(n, int) or n < 1:\n return '?'\n letter = chr(ord('a') + n - 1) if n <= 26 else str(n)\n
return letter.upper() if is_goldmine else letter\n",
"name": "num_to_alpha"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "function",
"content": "def wrap_text(text, width):\n \"\"\"Wrap string into a list of fixed-width segments.\"\"\"\n if not text:\n return
[\"\"]\n return [text[i:i+width] for i in range(0, len(text), width)]\n",
"name": "wrap_text"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "function",
"content": "def format_grid_code(race_info_list, wrap_width=4):\n \"\"\"\n Standardizes the formatting of race code strings
for the grid.\n Includes midpoint space for readability if length exceeds 5.\n\n Args:\n race_info_list: List of (race_num,
is_goldmine) tuples\n wrap_width: Width to wrap at\n \"\"\"\n if not race_info_list:\n return [\"\"]\n\n code =
\"\".join([num_to_alpha(n, gm) for n, gm in sorted(list(set(race_info_list)))])\n\n # Midpoint space logic for readability
(Project Convention)\n if len(code) > 5:\n mid = len(code) // 2\n code = code[:mid] + \" \" + code[mid:]\n\n return
wrap_text(code, wrap_width)\n",
"name": "format_grid_code"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "function",
"content": "def format_prediction_row(race: Race) -> str:\n \"\"\"Formats a single race prediction for the GHA Job Summary
table.\"\"\"\n metadata = getattr(race, 'metadata', {})\n\n st = race.start_time\n if isinstance(st, str):\n try: st =
datetime.fromisoformat(st.replace('Z', '+00:00'))\n except Exception: st = None\n date_str = st.strftime('%m/%d') if st else
'??/??'\n\n gold = '\u2705' if metadata.get('is_goldmine') else '\u2014'\n selection = metadata.get('selection_name') or
f\"#{metadata.get('selection_number', '?')}\"\n odds = metadata.get('predicted_2nd_fav_odds')\n odds_str = f\"{odds:.2f}\" if
odds else 'N/A'\n top5 = getattr(race, 'top_five_numbers', 'TBD')\n gap = metadata.get('1Gap2', 0.0)\n gap_str =
f\"{gap:.2f}\"\n\n payouts = []\n # Check both metadata and attributes for payouts\n for label in ('top1_place_payout',
'trifecta_payout', 'superfecta_payout'):\n val = metadata.get(label) or getattr(race, label, None)\n if val:\n display_label =
label.replace('_', ' ').title().replace('Top1 ', '')\n payouts.append(f\"{display_label}: ${float(val):.2f}\")\n\n payout_text
= ' | '.join(payouts) or 'Awaiting Results'\n return f\"| {date_str} | {race.venue} | {race.race_number} | {selection} |
{odds_str} | {gap_str} | {gold} | {top5} | {payout_text} |\"\n",
"name": "format_prediction_row"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "function",
"content": "def format_predictions_section(qualified_races: List[Race]) -> str:\n \"\"\"Generates the Predictions & Proof
section for the GHA Job Summary.\"\"\"\n lines = [\"### \ud83d\udd2e Fortuna Predictions & Proof\", \"\"]\n if not
qualified_races:\n lines.append(\"No Goldmine predictions available for this run.\")\n return \"\\n\".join(lines)\n\n now =
datetime.now(EASTERN)\n\n def get_mtp(r):\n st = r.start_time\n if isinstance(st, str):\n try:\n st =
datetime.fromisoformat(st.replace('Z', '+00:00'))\n except Exception:\n return 9999\n if st and st.tzinfo is None:\n st =
st.replace(tzinfo=EASTERN)\n return (st - now).total_seconds() / 60 if st else 9999\n\n # Sort by MTP ascending\n sorted_races
= sorted(qualified_races, key=get_mtp)\n # Take top 10 opportunities\n top_10 = sorted_races[:10]\n\n lines.extend([\n \"|
Date | Venue | Race# | Selection | Odds | Gap | Goldmine? | Pred Top 5 | Payout Proof |\",\n \"| --- | --- | --- | --- | --- |
--- | --- | --- | --- |\"\n ])\n for r in top_10:\n lines.append(format_prediction_row(r))\n return \"\\n\".join(lines)\n",
"name": "format_predictions_section"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "async_function",
"content": "async def format_proof_section(db: FortunaDB) -> str:\n \"\"\"Generates the Recent Audited Proof subsection for
the GHA Job Summary.\"\"\"\n lines = [\"\", \"#### \ud83d\udcb0 Recent Audited Proof\", \"\"]\n try:\n # First attempt to get
recent goldmines\n tips = await db.get_recent_audited_goldmines(limit=10)\n # Fallback to any audited tips if no goldmines
found\n if not tips:\n tips = await db.get_all_audited_tips()\n tips = tips[:10]\n\n if not tips:\n lines.append(\"Awaiting
race results; nothing audited yet.\")\n return \"\\n\".join(lines)\n\n lines.extend([\n \"| Verdict | Profit | Venue | R# |
Actual Top 5 | Actual 2nd Fav Odds | Payout Details |\",\n \"| :--- | :--- | :--- | :--- | :--- | :--- | :--- |\"\n ])\n for
tip in tips:\n payouts = []\n if tip.get('superfecta_payout'):\n payouts.append(f\"Superfecta
${tip['superfecta_payout']:.2f}\")\n if tip.get('trifecta_payout'):\n payouts.append(f\"Trifecta
${tip['trifecta_payout']:.2f}\")\n if tip.get('top1_place_payout'):\n payouts.append(f\"Place
${tip['top1_place_payout']:.2f}\")\n\n payout_text = ' / '.join(payouts) if payouts else 'No payout data'\n\n verdict =
tip.get(\"verdict\", \"?\")\n emoji = \"\u2705\" if verdict == \"CASHED\" else \"\u274c\" if verdict == \"BURNED\" else
\"\u26aa\"\n profit = tip.get('net_profit', 0.0)\n actual_odds = tip.get('actual_2nd_fav_odds')\n actual_odds_str =
f\"{actual_odds:.2f}\" if actual_odds else \"N/A\"\n\n lines.append(\n f\"| {emoji} {verdict} | ${profit:+.2f} |
{tip['venue']} | {tip['race_number']} | {tip.get('actual_top_5', 'N/A')} | {actual_odds_str} | {payout_text} |\"\n )\n except
Exception as e:\n lines.append(f\"Error generating audited proof: {e}\")\n\n return \"\\n\".join(lines)\n",
"name": "format_proof_section"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "function",
"content": "def build_harvest_table(summary: Dict[str, Any], title: str) -> str:\n \"\"\"Generates a harvest performance table
for the GHA Job Summary.\"\"\"\n lines = [f\"### {title}\", \"\"]\n if not summary:\n lines.extend([\n \"| Adapter | Races |
Max Odds | Status |\",\n \"| --- | --- | --- | --- |\",\n \"| N/A | 0 | 0.0 | \u26a0\ufe0f No harvest data |\"\n ])\n return
\"\\n\".join(lines)\n\n lines.extend([\n \"| Adapter | Races | Max Odds | Status |\",\n \"| --- | --- | --- | --- |\"\n ])\n\n
# Sort by Records Found (descending), then alphabetically\n def sort_key(item):\n adapter, data = item\n count =
data.get('count', 0) if isinstance(data, dict) else data\n return (-count, adapter)\n\n sorted_adapters =
sorted(summary.items(), key=sort_key)\n\n for adapter, data in sorted_adapters:\n if isinstance(data, dict):\n count =
data.get('count', 0)\n max_odds = data.get('max_odds', 0.0)\n else:\n count = data\n max_odds = 0.0\n\n status = '\u2705' if
count > 0 else '\u26a0\ufe0f No Data'\n lines.append(f\"| {adapter} | {count} | {max_odds:.1f} | {status} |\")\n return
\"\\n\".join(lines)\n",
"name": "build_harvest_table"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "function",
"content": "def format_artifact_links() -> str:\n \"\"\"Generates the report artifacts links for the GHA Job Summary.\"\"\"\n
return '\\n'.join([\n \"### \ud83d\udcc1 Report Artifacts\",\n \"\",\n \"- [Summary Grid](summary_grid.txt)\",\n \"- [Field
Matrix](field_matrix.txt)\",\n \"- [Goldmine Report](goldmine_report.txt)\",\n \"- [HTML Report](fortuna_report.html)\",\n \"-
[Analytics Log](analytics_report.txt)\"\n ])\n",
"name": "format_artifact_links"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "import",
"content": "from contextlib import contextmanager\n"
},
{
"type": "miscellaneous",
"content": "\n@contextmanager\n"
},
{
"type": "function",
"content": "def open_summary():\n \"\"\"Context manager for writing to GHA Job Summary with fallback to stdout.\"\"\"\n path =
os.environ.get('GITHUB_STEP_SUMMARY')\n if path:\n with open(path, 'a', encoding='utf-8') as f:\n yield f\n else:\n # Fallback
to stdout if not in GHA\n yield sys.stdout\n",
"name": "open_summary"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "function",
"content": "def write_job_summary(predictions_md: str, harvest_md: str, proof_md: str, artifacts_md: str) -> None:\n
\"\"\"Writes the consolidated sections to $GITHUB_STEP_SUMMARY using an efficient context manager.\"\"\"\n with open_summary()
as f:\n # Narrate the entire workflow\n summary = '\\n'.join([\n predictions_md,\n '',\n harvest_md,\n '',\n proof_md,\n '',\n
artifacts_md,\n ])\n try:\n f.write(summary + '\\n')\n except Exception:\n pass\n",
"name": "write_job_summary"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "function",
"content": "def get_db_path() -> str:\n \"\"\"Returns the path to the SQLite database, using AppData in frozen mode.\"\"\"\n
if is_frozen() and sys.platform == \"win32\":\n appdata = os.getenv('APPDATA')\n if appdata:\n db_dir = Path(appdata) /
\"Fortuna\"\n db_dir.mkdir(parents=True, exist_ok=True)\n return str(db_dir / \"fortuna.db\")\n\n return
os.environ.get(\"FORTUNA_DB_PATH\", \"fortuna.db\")\n",
"name": "get_db_path"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "class",
"content": "class FortunaDB:\n \"\"\"\n Thread-safe SQLite backend for Fortuna using the standard library.\n Handles
persistence for tips, predictions, and audit outcomes.\n \"\"\"\n def __init__(self, db_path: Optional[str] = None):\n
```

```
self.db_path = db_path or get_db_path()\n self._executor = ThreadPoolExecutor(max_workers=1)\n self._conn = None\n
self._conn_lock = threading.Lock()\n\n self._initialized = False\n self.logger =
structlog.get_logger(self.__class__.__name__)\n\n def _get_conn(self):\n with self._conn_lock:\n if not self._conn:\n
self._conn = sqlite3.connect(self.db_path, check_same_thread=False)\n self._conn.row_factory = sqlite3.Row\n # Enable WAL mode
for better concurrency\n self._conn.execute(\"PRAGMA journal_mode=WAL\")\n return self._conn\n\n @asynccontextmanager\n async
def get_connection(self):\n \"\"\"Returns an async context manager for a database connection.\"\"\"\n try:\n import
aiosqlite\n except ImportError:\n self.logger.error(\"aiosqlite not installed. Async database features will fail.\")\n
raise\n\n async with aiosqlite.connect(self.db_path) as conn:\n conn.row_factory = aiosqlite.Row\n yield conn\n\n async def
_run_in_executor(self, func, *args):\n loop = asyncio.get_running_loop()\n return await loop.run_in_executor(self._executor,
func, *args)\n\n async def initialize(self):\n \"\"\"Creates the database schema if it doesn't exist.\"\"\"\n if
self._initialized: return\n\n def _init():\n conn = self._get_conn()\n with conn:\n conn.execute(\"\"\"\n CREATE TABLE IF NOT
EXISTS schema_version (\n version INTEGER PRIMARY KEY,\n applied_at TEXT NOT NULL\n )\n \"\"\")\n conn.execute(\"\"\"\n CREATE
TABLE IF NOT EXISTS harvest_logs (\n id INTEGER PRIMARY KEY AUTOINCREMENT,\n timestamp TEXT NOT NULL,\n region TEXT,\n
adapter_name TEXT NOT NULL,\n race_count INTEGER NOT NULL,\n max_odds REAL\n )\n \"\"\")\n conn.execute(\"\"\"\n CREATE TABLE
IF NOT EXISTS tips (\n id INTEGER PRIMARY KEY AUTOINCREMENT,\n race_id TEXT NOT NULL,\n venue TEXT NOT NULL,\n race_number
INTEGER NOT NULL,\n discipline TEXT,\n start_time TEXT NOT NULL,\n report_date TEXT NOT NULL,\n is_goldmine INTEGER NOT
NULL,\n gap12 TEXT,\n top_five TEXT,\n selection_number INTEGER,\n selection_name TEXT,\n audit_completed INTEGER DEFAULT 0,\n
verdict TEXT,\n net_profit REAL,\n selection_position INTEGER,\n actual_top_5 TEXT,\n actual_2nd_fav_odds REAL,\n
trifecta_payout REAL,\n trifecta_combination TEXT,\n superfecta_payout REAL,\n superfecta_combination TEXT,\n
top1_place_payout REAL,\n top2_place_payout REAL,\n predicted_2nd_fav_odds REAL,\n audit_timestamp TEXT\n )\n \"\"\")\n #
Composite index for deduplication - changed to race_id only for better deduplication\n conn.execute(\"DROP INDEX IF EXISTS
idx_race_report\")\n\n # Cleanup potential duplicates before creating unique index (Memory Directive Fix)\n try:\n
self.logger.info(\"Cleaning up duplicate race_ids before indexing\")\n conn.execute(\"\"\"\n DELETE FROM tips\n WHERE id NOT
IN (\n SELECT MAX(id)\n FROM tips\n GROUP BY race_id\n )\n \"\"\")\n self.logger.info(\"Duplicates removed, creating unique
index\")\n conn.execute(\"CREATE UNIQUE INDEX IF NOT EXISTS idx_race_id ON tips (race_id)\")\n except Exception as e:\n
self.logger.error(\"Failed to cleanup or create unique index\", error=str(e))\n # If index exists but table has duplicates, we
might get IntegrityError\n # Just log it and continue - better than crashing the whole app\n # Composite index for audit
performance\n conn.execute(\"CREATE INDEX IF NOT EXISTS idx_audit_time ON tips (audit_completed, start_time)\")\n
conn.execute(\"CREATE INDEX IF NOT EXISTS idx_venue ON tips (venue)\")\n conn.execute(\"CREATE INDEX IF NOT EXISTS
idx_discipline ON tips (discipline)\")\n\n # Add missing columns for existing databases\n cursor = conn.execute(\"PRAGMA
table_info(tips)\")\n columns = [column[1] for column in cursor.fetchall()]\n if \"superfecta_payout\" not in columns:\n
conn.execute(\"ALTER TABLE tips ADD COLUMN superfecta_payout REAL\")\n if \"superfecta_combination\" not in columns:\n
conn.execute(\"ALTER TABLE tips ADD COLUMN superfecta_combination TEXT\")\n if \"top1_place_payout\" not in columns:\n
conn.execute(\"ALTER TABLE tips ADD COLUMN top1_place_payout REAL\")\n if \"top2_place_payout\" not in columns:\n
conn.execute(\"ALTER TABLE tips ADD COLUMN top2_place_payout REAL\")\n if \"discipline\" not in columns:\n
conn.execute(\"ALTER TABLE tips ADD COLUMN discipline TEXT\")\n if \"predicted_2nd_fav_odds\" not in columns:\n
conn.execute(\"ALTER TABLE tips ADD COLUMN predicted_2nd_fav_odds REAL\")\n if \"actual_2nd_fav_odds\" not in columns:\n
conn.execute(\"ALTER TABLE tips ADD COLUMN actual_2nd_fav_odds REAL\")\n if \"selection_name\" not in columns:\n
conn.execute(\"ALTER TABLE tips ADD COLUMN selection_name TEXT\")\n\n # Maintenance: Purge garbage data (Memory Directive
Fix)\n try:\n res = conn.execute(\"DELETE FROM tips WHERE selection_name = 'Runner 2' OR predicted_2nd_fav_odds IN (2.75)\")\n
if res.rowcount > 0:\n self.logger.info(\"Garbage data purged\", count=res.rowcount)\n except Exception as e:\n
self.logger.error(\"Failed to purge garbage data\", error=str(e))\n\n await self._run_in_executor(_init)\n\n # Track and
execute migrations based on schema version\n def _get_version():\n cursor = self._get_conn().execute(\"SELECT MAX(version)
FROM schema_version\")\n row = cursor.fetchone()\n return row[0] if row and row[0] is not None else 0\n\n current_version =
await self._run_in_executor(_get_version)\n\n if current_version < 2:\n await self.migrate_utc_to_eastern()\n def
_update_version():\n with self._get_conn() as conn:\n conn.execute(\"INSERT OR REPLACE INTO schema_version (version,
applied_at) VALUES (2, ?)\", (datetime.now(EASTERN).isoformat(),))\n await self._run_in_executor(_update_version)\n
self.logger.info(\"Schema migrated to version 2\")\n\n if current_version < 3:\n def _declutter():\n # Delete old records to
keep database lean (30-day retention cleanup)\n cutoff = (datetime.now(EASTERN) - timedelta(days=30)).isoformat()\n with
self._get_conn() as conn:\n cursor = conn.execute(\"DELETE FROM tips WHERE report_date < ?\", (cutoff,))\n
self.logger.info(\"Database decluttered (30-day retention cleanup)\", deleted_count=cursor.rowcount)\n conn.execute(\"INSERT
OR REPLACE INTO schema_version (version, applied_at) VALUES (3, ?)\", (datetime.now(EASTERN).isoformat(),))\n await
self._run_in_executor(_declutter)\n self.logger.info(\"Schema migrated to version 3\")\n\n if current_version < 4:\n #
Migration to version 4: Housekeeping & Long-term retention.\n # 1. Clear the tips table for a fresh start as requested by
JB.\n # 2. Historical retention is now enabled (auto-cleanup removed from future migrations).\n def _housekeeping():\n
self.logger.warning(\"Applying destructive migration: Clearing all historical tips for version 4 fresh start.\")\n with
self._get_conn() as conn:\n conn.execute(\"DELETE FROM tips\")\n conn.execute(\"INSERT OR REPLACE INTO schema_version
(version, applied_at) VALUES (4, ?)\", (datetime.now(EASTERN).isoformat(),))\n await self._run_in_executor(_housekeeping)\n
self.logger.info(\"Schema migrated to version 4 (Housekeeping complete, long-term retention enabled)\")\n\n self._initialized
= True\n self.logger.info(\"Database initialized\", path=self.db_path, schema_version=max(current_version, 4))\n\n async def
migrate_utc_to_eastern(self) -> None:\n \"\"\"Migrates existing database records from UTC to US Eastern Time.\"\"\"\n def
_migrate():\n conn = self._get_conn()\n cursor = conn.execute(\"\"\"\n SELECT id, start_time, report_date, audit_timestamp
FROM tips\n WHERE start_time LIKE '%+00:00' OR start_time LIKE '%Z'\n OR report_date LIKE '%+00:00' OR report_date LIKE '%Z'\n
OR audit_timestamp LIKE '%+00:00' OR audit_timestamp LIKE '%Z'\n \"\"\")\n rows = cursor.fetchall()\n if not rows: return\n\n
total = len(rows)\n self.logger.info(\"Migrating legacy UTC timestamps to Eastern\", count=total)\n converted = 0\n errors =
0\n\n # Process in chunks of 1000 for safety (Memory Directive Fix)\n for i in range(0, total, 1000):\n chunk =
rows[i:i+1000]\n with conn:\n for row in chunk:\n updates = {}\n for col in [\"start_time\", \"report_date\",
\"audit_timestamp\"]:\n if col not in row.keys(): continue\n val = row[col]\n if val:\n try:\n dt =
datetime.fromisoformat(val.replace(\"Z\", \"+00:00\"))\n dt_eastern = ensure_eastern(dt)\n updates[col] =
dt_eastern.isoformat()\n except Exception: pass\n if updates:\n try:\n set_clause = \", \".join([f\"{k} = ?\" for k in
updates.keys()])\n conn.execute(f\"UPDATE tips SET {set_clause} WHERE id = ?\", (*updates.values(), row[\"id\"]))\n converted
+= 1\n except Exception as e:\n errors += 1\n self.logger.warning(\"Failed to migrate row\", row_id=row[\"id\"],
error=str(e))\n self.logger.info(\"Migration progress\", processed=min(i + 1000, total), total=total)\n\n
self.logger.info(\"Migration complete\", total=total, converted=converted, errors=errors)\n await
self._run_in_executor(_migrate)\n\n async def log_harvest(self, harvest_summary: Dict[str, Any], region: Optional[str] =
None):\n \"\"\"Logs harvest performance metrics to the database.\"\"\"\n if not self._initialized: await self.initialize()\n\n
def _log():\n conn = self._get_conn()\n now = datetime.now(EASTERN).isoformat()\n to_insert = []\n for adapter, data in
harvest_summary.items():\n if isinstance(data, dict):\n count = data.get(\"count\", 0)\n max_odds = data.get(\"max_odds\",
0.0)\n else:\n count = data\n max_odds = 0.0\n\n to_insert.append((now, region, adapter, count, max_odds))\n\n if to_insert:\n
with conn:\n conn.executemany(\"\"\"\n INSERT INTO harvest_logs (timestamp, region, adapter_name, race_count, max_odds)\n
VALUES (?, ?, ?, ?, ?)\n \"\"\", to_insert)\n\n await self._run_in_executor(_log)\n\n async def get_adapter_scores(self, days:
int = 30) -> Dict[str, float]:\n \"\"\"Calculates historical performance scores for each adapter.\"\"\"\n if not
self._initialized: await self.initialize()\n\n def _get():\n conn = self._get_conn()\n cutoff = (datetime.now(EASTERN) -
timedelta(days=days)).isoformat()\n cursor = conn.execute(\"\"\"\n SELECT adapter_name,\n AVG(race_count) as avg_count,\n
AVG(max_odds) as avg_max_odds\n FROM harvest_logs\n WHERE timestamp > ?\n GROUP BY adapter_name\n \"\"\", (cutoff,))\n\n
```

```
scores = {}\n for row in cursor.fetchall():\n # Heuristic: Score = Avg Race Count + (Avg Max Odds * 2)\n # This prioritizes
adapters that find races and high longshots\n scores[row[\"adapter_name\"]] = (row[\"avg_count\"] or 0) +
((row[\"avg_max_odds\"] or 0) * 2)\n return scores\n\n return await self._run_in_executor(_get)\n\n async def log_tips(self,
tips: List[Dict[str, Any]], dedup_window_hours: int = 12):\n \"\"\"Logs new tips to the database with batch
deduplication.\"\"\"\n if not self._initialized: await self.initialize()\n\n def _log():\n conn = self._get_conn()\n now =
datetime.now(EASTERN)\n\n # Batch check for recently logged tips to avoid redundant entries\n race_ids = [t.get(\"race_id\")
for t in tips if t.get(\"race_id\")]\n if not race_ids: return\n\n placeholders = \",\".join([\"?\"] * len(race_ids))\n\n #
Use a more absolute check to ensure distinct races across all time\n cursor = conn.execute(\n f\"SELECT race_id FROM tips
WHERE race_id IN ({placeholders})\",\n (*race_ids,)\n )\n already_logged = {row[\"race_id\"] for row in cursor.fetchall()}\n\n
to_insert = []\n for tip in tips:\n rid = tip.get(\"race_id\")\n if rid and rid not in already_logged:\n report_date =
tip.get(\"report_date\") or now.isoformat()\n to_insert.append((\n rid, tip.get(\"venue\"), tip.get(\"race_number\"),\n
tip.get(\"discipline\"), tip.get(\"start_time\"), report_date,\n 1 if tip.get(\"is_goldmine\") else 0,\n
str(tip.get(\"1Gap2\", 0.0)),\n tip.get(\"top_five\"), tip.get(\"selection_number\"), tip.get(\"selection_name\"),\n
float(tip.get(\"predicted_2nd_fav_odds\")) if tip.get(\"predicted_2nd_fav_odds\") is not None else None\n ))\n
already_logged.add(rid) # Avoid duplicates within the same batch\n\n if to_insert:\n with conn:\n conn.executemany(\"\"\"\n
INSERT OR IGNORE INTO tips (\n race_id, venue, race_number, discipline, start_time, report_date,\n is_goldmine, gap12,
top_five, selection_number, selection_name, predicted_2nd_fav_odds\n ) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)\n \"\"\",
to_insert)\n self.logger.info(\"Hot tips batch logged\", count=len(to_insert))\n\n await self._run_in_executor(_log)\n\n async
def get_unverified_tips(self, lookback_hours: int = 48) -> List[Dict[str, Any]]:\n \"\"\"Returns tips that haven't been
audited yet but have likely finished.\"\"\"\n if not self._initialized: await self.initialize()\n\n def _get():\n conn =
self._get_conn()\n now = datetime.now(EASTERN)\n cutoff = (now - timedelta(hours=lookback_hours)).isoformat()\n\n cursor =
conn.execute(\n \"\"\"SELECT * FROM tips\n WHERE audit_completed = 0\n AND report_date > ?\n AND start_time < ?\"\"\",\n
(cutoff, now.isoformat())\n )\n return [dict(row) for row in cursor.fetchall()]\n return await self._run_in_executor(_get)\n\n
async def get_recent_tips(self, limit: int = 20) -> List[Dict[str, Any]]:\n \"\"\"Returns the most recent tips regardless of
audit status, ordered by discovery time.\"\"\"\n if not self._initialized: await self.initialize()\n def _get():\n # Use ID
DESC to show most recently discovered tips first\n cursor = self._get_conn().execute(\n \"SELECT * FROM tips ORDER BY id DESC
LIMIT ?\",\n (limit,)\n )\n return [dict(row) for row in cursor.fetchall()]\n return await self._run_in_executor(_get)\n\n
async def update_audit_result(self, race_id: str, outcome: Dict[str, Any]):\n \"\"\"Updates a single tip with its audit
outcome.\"\"\"\n if not self._initialized: await self.initialize()\n\n def _update():\n conn = self._get_conn()\n with conn:\n
conn.execute(\"\"\"\n UPDATE tips SET\n audit_completed = 1,\n verdict = ?,\n net_profit = ?,\n selection_position = ?,\n
actual_top_5 = ?,\n actual_2nd_fav_odds = ?,\n trifecta_payout = ?,\n trifecta_combination = ?,\n superfecta_payout = ?,\n
superfecta_combination = ?,\n top1_place_payout = ?,\n top2_place_payout = ?,\n audit_timestamp = ?\n WHERE id = (\n SELECT id
FROM tips\n WHERE race_id = ? AND audit_completed = 0\n LIMIT 1\n )\n \"\"\", (\n outcome.get(\"verdict\"),
outcome.get(\"net_profit\"),\n outcome.get(\"selection_position\"), outcome.get(\"actual_top_5\"),\n
outcome.get(\"actual_2nd_fav_odds\"), outcome.get(\"trifecta_payout\"),\n outcome.get(\"trifecta_combination\"),\n
outcome.get(\"superfecta_payout\"),\n outcome.get(\"superfecta_combination\"),\n outcome.get(\"top1_place_payout\"),\n
outcome.get(\"top2_place_payout\"),\n datetime.now(EASTERN).isoformat(),\n race_id\n ))\n await
self._run_in_executor(_update)\n\n async def update_audit_results_batch(self, outcomes: List[Tuple[str, Dict[str, Any]]]):\n
\"\"\"Updates multiple tips with their audit outcomes in a single transaction.\"\"\"\n if not outcomes: return\n if not
self._initialized: await self.initialize()\n\n def _update():\n conn = self._get_conn()\n with conn:\n for race_id, outcome in
outcomes:\n conn.execute(\"\"\"\n UPDATE tips SET\n audit_completed = 1,\n verdict = ?,\n net_profit = ?,\n selection_position
= ?,\n actual_top_5 = ?,\n actual_2nd_fav_odds = ?,\n trifecta_payout = ?,\n trifecta_combination = ?,\n superfecta_payout =
?,\n superfecta_combination = ?,\n top1_place_payout = ?,\n top2_place_payout = ?,\n audit_timestamp = ?\n WHERE id = (\n
SELECT id FROM tips\n WHERE race_id = ? AND audit_completed = 0\n LIMIT 1\n )\n \"\"\", (\n outcome.get(\"verdict\"),
outcome.get(\"net_profit\"),\n outcome.get(\"selection_position\"), outcome.get(\"actual_top_5\"),\n
outcome.get(\"actual_2nd_fav_odds\"), outcome.get(\"trifecta_payout\"),\n outcome.get(\"trifecta_combination\"),\n
outcome.get(\"superfecta_payout\"),\n outcome.get(\"superfecta_combination\"),\n outcome.get(\"top1_place_payout\"),\n
outcome.get(\"top2_place_payout\"),\n outcome.get(\"audit_timestamp\"),\n race_id\n ))\n await
self._run_in_executor(_update)\n\n async def get_all_audited_tips(self) -> List[Dict[str, Any]]:\n \"\"\"Returns all audited
tips for reporting.\"\"\"\n if not self._initialized: await self.initialize()\n def _get():\n cursor =
self._get_conn().execute(\n \"SELECT * FROM tips WHERE audit_completed = 1 ORDER BY start_time DESC\"\n )\n return [dict(row)
for row in cursor.fetchall()]\n return await self._run_in_executor(_get)\n\n async def get_recent_audited_goldmines(self,
limit: int = 15) -> List[Dict[str, Any]]:\n \"\"\"Returns recent successfully audited goldmine tips.\"\"\"\n if not
self._initialized: await self.initialize()\n def _get():\n cursor = self._get_conn().execute(\n \"SELECT * FROM tips WHERE
audit_completed = 1 AND is_goldmine = 1 ORDER BY start_time DESC LIMIT ?\",\n (limit,)\n )\n return [dict(row) for row in
cursor.fetchall()]\n return await self._run_in_executor(_get)\n\n async def clear_all_tips(self):\n \"\"\"Wipes all records
from the tips table.\"\"\"\n if not self._initialized: await self.initialize()\n def _clear():\n conn = self._get_conn()\n
with conn:\n conn.execute(\"DELETE FROM tips\")\n conn.execute(\"VACUUM\")\n self.logger.info(\"Database cleared (all tips
deleted)\")\n await self._run_in_executor(_clear)\n\n async def migrate_from_json(self, json_path: str =
\"hot_tips_db.json\"):\n \"\"\"Migrates data from existing JSON file to SQLite with detailed error logging.\"\"\"\n path =
Path(json_path)\n if not path.exists(): return\n try:\n with open(path, \"r\") as f:\n data = json.load(f)\n if not
isinstance(data, list): return\n self.logger.info(\"Migrating data from JSON\", count=len(data))\n if not self._initialized:
await self.initialize()\n\n def _migrate():\n conn = self._get_conn()\n success_count = 0\n for entry in data:\n try:\n with
conn:\n conn.execute(\"\"\"\n INSERT OR IGNORE INTO tips (\n race_id, venue, race_number, start_time, report_date,\n
is_goldmine, gap12, top_five, selection_number,\n audit_completed, verdict, net_profit, selection_position,\n actual_top_5,
actual_2nd_fav_odds, trifecta_payout,\n trifecta_combination, superfecta_payout,\n superfecta_combination,
top1_place_payout,\n top2_place_payout, audit_timestamp\n ) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,
?, ?, ?)\n \"\"\", (\n entry.get(\"race_id\"), entry.get(\"venue\"), entry.get(\"race_number\"),\n entry.get(\"start_time\"),
entry.get(\"report_date\"),\n 1 if entry.get(\"is_goldmine\") else 0, str(entry.get(\"1Gap2\", 0.0)),\n
entry.get(\"top_five\"), entry.get(\"selection_number\"),\n 1 if entry.get(\"audit_completed\") else 0,
entry.get(\"verdict\"),\n entry.get(\"net_profit\"), entry.get(\"selection_position\"),\n entry.get(\"actual_top_5\"),
entry.get(\"actual_2nd_fav_odds\"),\n entry.get(\"trifecta_payout\"), entry.get(\"trifecta_combination\"),\n
entry.get(\"superfecta_payout\"), entry.get(\"superfecta_combination\"),\n entry.get(\"top1_place_payout\"),
entry.get(\"top2_place_payout\"),\n entry.get(\"audit_timestamp\")\n ))\n success_count += 1\n except Exception as e:\n
self.logger.error(\"Failed to migrate entry\", race_id=entry.get(\"race_id\"), error=str(e))\n return success_count\n\n count
= await self._run_in_executor(_migrate)\n self.logger.info(\"Migration complete\", successful=count)\n except Exception as
e:\n self.logger.error(\"Migration failed\", error=str(e))\n\n async def close(self):\n def _close():\n if self._conn:\n
self._conn.close()\n self._conn = None\n\n await self._run_in_executor(_close)\n self._executor.shutdown(wait=True)\n",
"name": "FortunaDB"
}
]
}
```