# Section 1: Executive Summary — Revised Structure

---

## 1.1 What It Is

- Definition: An AI-powered personality profiling system that uses the Seven Deadly Sins framework to convert natural language responses into quantified trait profiles
- Core mechanism: Users answer 6 scenario-based questions; Gemini AI parses their open-ended responses into scores across seven dimensions

---

## 1.2 The Problem It Solves

- Joel et al. (2017) demonstrated that machine learning could not predict relationship-specific attraction from pre-meeting profile data — even with 100+ self-report measures
- The study found algorithms could predict "actor variance" (who tends to like others) and "partner variance" (who tends to be liked), but not "relationship variance" (which two specific people will click)
- Implication: Traditional personality matching is fundamentally limited — you cannot shortcut the process of meeting someone

---

## 1.3 Harmonia's Response: Focus on What Can Be Measured

- Rather than attempting to predict unpredictable chemistry, the system focuses on:
    - **Perceived similarity** — research shows this matters more than actual similarity (Montoya et al. 2008 meta-analysis found perceived similarity affects attraction throughout relationship development, even when that perception is wrong)
    - **Positive overlap detection** — highlighting shared traits rather than differences or complementary opposites
- The system operates invisibly in the backend; users never see raw personality scores

---

## 1.4 The "Perceived Similarity" Philosophy

- Research basis: Partners who are perceived as similar report greater feelings of being understood and validated (Murray, Holmes, & Griffin, 1996)
- The system answers: "If these two people met and talked about their personalities, how much would they feel alike?"

- Designed to create an "astrology effect" — post-match revelations that reinforce perceived connection
- Only surfaces what users share, never differences

---

## 1.5 Why Scenario-Based Assessment

- Traditional self-report questionnaires are prone to social desirability bias
- Scenario-based assessment (similar to Situational Judgment Tests) extracts authentic behavioral tendencies by asking how users would respond to real-life situations
- The 6 questions cover distinct life domains:
    1. **The Group Dinner Check** — resource sharing, social dynamics
    2. **The Unexpected Expense —** stress response, financial reasoning
    3. **The Weekend Off** — leisure priorities, energy restoration
    4. **The Unequal Split** — conflict handling, fairness expectations
    5. **The Friend Crisis —** loyalty trade-offs, relationship prioritisation
    6. **The Feedback Received** — ego resilience, growth orientation
- Allows AI parsing to detect unconscious patterns, values hierarchies, and conflict resolution styles that users may not explicitly recognise in themselves

---

## 1.6 High-Level Flow (one paragraph)

- User answers 6 scenario-based questions (one per block) → Gemini parses each response into Seven Deadly Sins trait scores with confidence levels → Individual scores aggregate into a personality profile → Profiles are compared against other users using a positive-overlap algorithm → Similarity score feeds into the ranking system alongside visual and genetic components

# Section 2: Input — Felix's 6 Questions

---

## 2.1 Overview of the Question Set

The personality assessment uses a fixed set of 6 scenario-based questions, developed by Felix Noble (Harmonia's Scientific Advisor, Masters in Neuroscience from Goldsmiths). Each question targets a distinct life domain and is designed to activate multiple personality dimensions simultaneously.

Every user answers the same 6 questions in the same order. This is not randomised — consistency is essential for the system to compare profiles meaningfully across users. If User A answered different questions than User B, their Seven Deadly Sins profiles would be derived from different stimuli, making similarity calculations unreliable.

The questions evolved from an earlier 28-question framework (organised into 7 blocks of 4 questions each, with randomised selection within blocks). The streamlined 6-question format reduces cognitive burden while maintaining sufficient signal for AI parsing.

---

## 2.2 Why Scenario-Based (Not Self-Report)

Traditional personality questionnaires rely on self-report: users rate how much they agree with statements like "I am outgoing" or "I get stressed easily." This approach has well-documented limitations.

**The faking problem.** Research estimates that 30-63% of job applicants distort their responses on self-report personality tests. Up to 50% admit to exaggerating positive qualities, while 60% admit to de-emphasising negative traits. In a dating context — where users are motivated to appear attractive — this distortion is likely even higher.

**Social desirability bias.** When questions are transparent ("Do you get angry easily?"), users can identify the "right" answer and respond accordingly. Face-valid inventories make it easy to game the system.

**Situational Judgment Tests offer a solution.** SJTs are low-fidelity simulations that present realistic scenarios and assess how individuals would respond. Research demonstrates that SJTs are less prone to faking than traditional self-report measures (Motowidlo et al. 1990). The key insight is behavioural consistency theory: how someone handles a hypothetical dinner bill likely predicts how they handle similar resource-sharing situations in actual relationships.

SJTs have demonstrated consistent criterion-related validity across work settings for decades (Christian et al. 2010 meta-analysis). They assess compound knowledge, skills, and abilities that don't fall cleanly into cognitive ability or personality categories — exactly the kind of nuanced behavioural tendencies relevant to relationship compatibility.

By asking "What would you do if..." rather than "Are you a generous person?", scenario-based questions extract authentic behavioural tendencies rather than curated self-descriptions.

---

## 2.3 Why Open-Ended (Not Multiple Choice)

The questions require free-text responses rather than selecting from pre-defined options. This design choice is grounded in both psychological research and practical AI capabilities.

**Multiple choice limits expression.** When response options are pre-defined, users are constrained to the researcher's framing. They cannot express nuance, mixed feelings, or reasoning that falls outside the provided options. Forced-choice formats limit participants' responses and may not capture the full range of how they would actually behave.

**Open-ended responses capture richer signal.** Research on language-based personality assessment shows that answers to open-ended questions are more likely to contain a multifaceted, comprehensive portrayal of respondents, including the order, interactions, and context of events. Unlike self-report questionnaires that limit responses to predefined options, text responses capture a broader range of emotions, thought patterns, and behavioural indicators.

**AI can parse what humans cannot scale.** Traditional open-ended questionnaires required manual coding — expensive and slow. Gemini AI can analyse natural language responses at scale, detecting implicit trait signals that fixed options would miss: tone, justification patterns, values hierarchies, contradictions, defence mechanisms.

**Harder to game.** With multiple choice, users can reverse-engineer which option sounds most attractive. With open-ended responses, there's no "right answer" to identify. Users must construct their own response, revealing authentic reasoning patterns even when trying to present well.

**Research supports this approach.** A 2024 study on predicting neuroticism using open-ended questions found that language-based personality assessment using natural language processing could effectively predict personality traits from free-text responses, with the advantage that respondents reveal various aspects of themselves in a relatively less constrained manner.

---

## 2.4 The 6 Questions — Domain Coverage

Each question is designed to activate multiple personality dimensions (sins) depending on how the user responds. The questions cover distinct life domains that most adults have encountered or can readily imagine.

---

**Question 1: The Group Dinner Check**

> *"The bill arrives at a group dinner. Everyone contributed differently to the total. What's your approach?"*

**Domain:** Resource sharing, fairness, social harmony vs. self-interest

**Why this question:** Money is a universal pressure point in relationships. How someone handles bill-splitting reveals attitudes toward fairness, conflict avoidance, generosity, and social image. A user who insists on itemised splitting may score differently on Greed and Wrath than one who suggests splitting evenly "to keep things simple."

**Potential sin activations:**

- Greed (materialism, self-interest vs. generosity)
- Wrath (willingness to confront unfairness vs. conflict avoidance)
- Pride (concern with how they're perceived by the group)
- Sloth (preferring the easy solution vs. doing the math)

---

**Question 2: The Unexpected Expense**

> *"Your car needs a $1,200 repair you didn't budget for. Walk me through your thought process."*

**Domain:** Stress response, financial reasoning, emotional regulation

**Why this question:** Unexpected financial stress reveals coping mechanisms and emotional regulation patterns. Some users will catastrophise; others will immediately problem-solve. The question asks for "thought process" rather than just "what would you do" — inviting users to reveal their internal reasoning.

**Potential sin activations:**

- Wrath (frustration, anger at the situation)
- Sloth (avoidance vs. proactive problem-solving)
- Gluttony (impulse to spend despite budget strain)
- Pride (shame about financial situation, reluctance to ask for help)

---

**Question 3: The Weekend Off**

> *"You have a completely free weekend with no obligations. How do you spend it?"*

**Domain:** Leisure priorities, introversion/extroversion, self-care patterns

**Why this question:** How someone chooses to spend unstructured time reveals core values and energy patterns. Social vs. solitary activities, productive vs. restorative choices, planned vs. spontaneous approaches — all emerge naturally.

**Potential sin activations:**

- Lust (novelty-seeking, spontaneity, adventure)
- Sloth (rest and recovery vs. productivity)
- Gluttony (indulgence, treating oneself)
- Pride (activities that enhance status or image)
- Envy (comparison to how others spend their time)

---

**Question 4: The Unequal Split**

> *"You're working on a group project where one person is contributing significantly less than everyone else. How do you handle this?"*

**Domain:** Conflict handling, fairness expectations, assertiveness

**Why this question:** Group dynamics reveal interpersonal style under pressure. Does the user confront directly? Compensate silently? Escalate to authority? Passive-aggressively hint? The scenario is universal (work, school, community) and emotionally charged.

**Potential sin activations:**

- Wrath (confrontation style, anger at unfairness)
- Pride (ego involvement in the project's success)
- Sloth (willingness to do extra work vs. demanding accountability)
- Envy (resentment toward the underperformer)

---

**Question 5: The Friend Crisis**

> *"Your best friend calls with an emergency the same night as a planned date you've been looking forward to. What's your thinking process?"*

**Domain:** Loyalty trade-offs, relationship prioritisation, decision-making under competing demands

**Why this question:** Competing obligations force users to reveal their values hierarchy. The question explicitly asks for "thinking process" — not just the outcome, but how they weigh the decision. Users who immediately prioritise the friend reveal different patterns than those who probe the nature of the "emergency."

**Potential sin activations:**

- Lust (anticipation of the date, disappointment at missing it)
- Pride (how they want to be seen by friend vs. date)
- Sloth (path of least resistance)
- Wrath (frustration at the timing)
- Greed (self-interest vs. others' needs)

---

**Question 6: The Feedback Received**

> *"Someone you respect gives you critical feedback about a blind spot they've noticed in your behavior. How do you process and respond to this?"*

**Domain:** Ego resilience, growth orientation, defensiveness vs. openness

**Why this question:** Response to criticism is a fundamental relationship predictor. Defensive reactions, dismissal, genuine reflection, excessive self-criticism — each pattern signals different personality structures. The framing ("someone you respect") prevents easy dismissal of the feedback source.

**Potential sin activations:**

- Pride (ego threat, defensiveness, shame)
- Wrath (anger at being criticised)
- Sloth (willingness to change vs. avoidance)
- Envy (comparison to the person giving feedback)

---

## 2.5 Design Principles Behind the Questions

The 6 questions share common design principles derived from both SJT research and Harmonia's specific needs:

**Universality.** The situations are ones most adults have encountered or can vividly imagine. No specialised knowledge is required. A university student and a 45-year-old professional can both engage meaningfully with "the bill arrives at a group dinner."

**Ambiguity.** There is no objectively correct answer. Splitting the bill evenly is not "better" than itemising; helping a friend is not "better" than keeping the date. Responses reveal values and priorities, not knowledge or competence.

**Multi-trait activation.** Each question can surface multiple sins depending on how the user responds. "The Group Dinner Check" might reveal Greed in one user, Wrath in another, and Sloth in a third — or a combination. This efficiency allows 6 questions to generate a full 7-dimensional profile.

**Behavioural consistency.** The scenarios are designed as proxies for relationship-relevant behaviour. How someone handles a dinner bill predicts how they'll handle shared expenses in a relationship. How they respond to criticism predicts how they'll handle conflict with a partner.

**Reasoning exposure.** Several questions explicitly ask for "thought process" or "how do you process" — inviting users to reveal internal reasoning, not just final decisions. This gives Gemini richer material to parse.

---

## 2.6 Response Constraints

Users must provide responses within defined boundaries:

| Constraint | Value | Rationale |
|---|---|---|
| **Minimum** | 25 words | Ensures sufficient signal for AI parsing; prevents single-sentence non-answers |
| **Maximum** | 150 words | Prevents rambling; keeps cognitive load manageable; research shows dropout increases with open-ended length |
| **Format** | Free-text | No pre-set options; no character restrictions beyond word count |

**Why these limits?**

The 25-word minimum ensures responses contain enough linguistic signal for Gemini to extract personality indicators. A response like "I'd split it evenly" (5 words) provides almost no signal about underlying values or reasoning.

The 150-word maximum prevents fatigue and keeps responses focused. Research on open-ended survey questions shows that dropout increases when participants face lengthy text-entry requirements. The limit also ensures all users provide roughly comparable amounts of information, preventing one verbose user from having an outsized profile relative to a concise one.

**Validation in the UI:**

- Real-time word count displayed as user types
- Colour-coded feedback (red if under minimum, yellow if approaching maximum)
- Submission blocked if constraints not met
- Progress saved to localStorage every 30 seconds to prevent loss

---

## 2.7 Why Not Randomised

Earlier iterations of the questionnaire used randomisation: 28 questions organised into 7 blocks, with 2 questions randomly selected from each block per user. This approach was abandoned for the 6-question format.

**The comparability problem.** If User A answers questions 1, 3, and 5, while User B answers questions 2, 4, and 6, their resulting Seven Sins profiles are derived from different stimuli. Comparing these profiles assumes the questions are interchangeable measures of the same underlying traits — an assumption that requires validation.

**The aggregation problem.** With randomised selection, each user's profile is an average across different question subsets. This introduces noise: observed differences between users might reflect question differences rather than personality differences.

**The solution: fixed questions.** By having every user answer the same 6 questions in the same order, the system ensures:

- Direct comparability: User A's response to "The Dinner Check" can be compared to User B's response to the same question
- Consistent stimuli: All profiles are derived from identical inputs
- Simpler validation: The system can be tested against the specific 6 questions rather than all possible combinations

The trade-off is reduced coverage (6 questions vs. 28) — but the streamlined set was designed to maximise multi-trait activation per question, maintaining adequate coverage of all seven sins.

---

## 2.8 Summary

Felix's 6 questions represent a distillation of scenario-based personality assessment principles into a format optimised for AI parsing and relationship compatibility prediction:

- **Fixed, not randomised** — ensures comparability across users
- **Scenario-based, not self-report** — reduces faking, extracts authentic behaviour
- **Open-ended, not multiple choice** — enables nuanced signal extraction via AI
- **Constrained length (25-150 words)** — balances signal richness with completion rates
- **Multi-trait activation** — each question can reveal multiple sins depending on response

The questions do not measure personality directly. They generate natural language responses that Gemini then parses into quantified trait profiles — the subject of Section 3.

## Section 3: The Gemini Parsing Architecture

---

### 3.1 Overview: What Gemini Does

Gemini serves as Harmonia's psychological parsing engine. For each user, Gemini receives the raw text of their six scenario responses and transforms them into structured personality data—numerical scores (-5 to +5), confidence ratings (0 to 1), and direct evidence quotes for each of the seven sins.

This is a text-to-structure translation task. Gemini receives one question-response pair at a time, extracts personality signals, and returns structured JSON. It operates statelessly—no memory between calls, no learning from previous users.

**Critical Context from Research:**

Recent validation studies (Zhu et al. 2025, AAAI) reveal that LLM-based personality inference from text achieves, at best:

| Metric | Best Achieved |
|---|---|
| Pearson correlation r | < 0.26 |
| Mean Absolute Error (1-5 scale) | 0.69-1.2 |
| Single-response reliability (α) | < 0.50 |

This means Harmonia's personality component should be framed as a **personality indicator** rather than a diagnostic assessment. The system captures self-presentation style and communication patterns—useful for perceived similarity matching—but not deep psychological truth.

**Harmonia's 6-Question Design:**

Research recommends 3-5 responses with 150+ words each for borderline-acceptable reliability ($\alpha$ ~0.55-0.65). Harmonia's design—6 fixed questions with 25-150 word responses—operates at the minimum viable threshold:

| Design | Expected Reliability |
|---|---|
| Single response | $\alpha < 0.50$ (unreliable) |
| 3-5 responses, 150+ words each | $\alpha$ ~0.55-0.65 (borderline) |
| **Harmonia: 6 responses, 25-150 words** | $\alpha$ ~0.50-0.60 (borderline-low) |

The fixed (non-randomised) question set means no reverse-coded items to detect inconsistent responding, and less trait-space coverage than randomised sampling. This makes multi-observer consensus and discrepancy detection critical—Harmonia must extract maximum signal from limited data.

---

## 3.2 Model Selection and Fallback Chain

**Model Priority (January 2026):**

| Priority | Model | Rationale |
|---|---|---|
| Primary | `gemini-3-pro-preview` | Maximum reasoning depth for nuanced interpretation |
| Fallback | `gemini-3-flash-preview` | Pro-level intelligence at lower latency |

| | | |
|---|---|---|
| Stable | `gemini-2.5-flash` | Production-stable if 3.x previews fail |

**Why Pro First:**

Research shows larger models don't dramatically improve personality inference accuracy. However, Pro's deeper semantic representation may capture subtle linguistic patterns that inform the multi-observer consensus. For a dating platform where match quality affects retention, the marginal improvement justifies the cost.

**Fallback Logic:**

python

```python
model_chain = [
    'gemini-3-pro-preview',
    'gemini-3-flash-preview',
    'gemini-2.5-flash'
]

for model_name in model_chain:
    try:
        response = call_model(model_name, prompt)
        return parse_response(response)
    except (RateLimitError, TimeoutError, QuotaError):
        continue

return default_neutral_scores()
```

---

### 3.3 Why Chain-of-Thought Doesn't Help

**The Research Finding:**

Zhu et al. (2025) tested Chain-of-Thought prompting against zero-shot on clinical interview data (n=518 with validated BFI-10 ground truth):

| Method | Conscientiousness r | Neuroticism r |
|--------|---------------------|---------------|
| Zero-shot | 0.250 | 0.065 |
| Chain-of-Thought | 0.236 | -0.009 |

CoT **degraded** performance, particularly for Neuroticism. The reason: personality inference relies on latent semantic representation, not explicit reasoning. Forcing the model to "think step by step" about linguistic patterns it has no training signal to interpret introduces noise rather than clarity.

**Implication for Harmonia:**

A prompt asking Gemini to reason through trait signals would be counterproductive. Instead, Harmonia uses **trait-specific decomposition**—focused prompts for each sin with anchored examples rather than open-ended reasoning.

---

### 3.4 Trait-Specific Decomposition

Rather than one prompt asking for all seven sins simultaneously, Harmonia uses separate prompts for each trait with scale anchors.

**Example: Wrath-Specific Prompt**

```
Analyze this response for WRATH signals only.

Wrath Scale (-5 to +5):
-5 = Extreme conflict avoidance: "I'd never confront anyone, even if wronged"
-3 = Strong patience: "I take a breath and try to understand their perspective"
 0 = Neutral: No clear signal about conflict handling
+3 = Quick to frustration: "That would really annoy me"
+5 = Explosive anger: "I'd lose my temper and tell them exactly what I think"

Question: "{question_text}"
Response: "{response_text}"

Return JSON:
{"score": <-5 to +5>, "confidence": <0.0 to 1.0>, "evidence": "<quote or empty>"}
```

**Why This Works Better:**

1. **Focused attention** — Model concentrates on one trait without cross-contamination

2. **Anchored scoring** — Concrete examples at scale endpoints reduce drift

3. **Parallel execution** — Seven calls run simultaneously; no latency penalty

4. **Independent confidence** — Each trait gets calibrated uncertainty

**Trait Anchors:**

| Sin | -5 Anchor | +5 Anchor |
|-----|-----------|-----------|
| Greed | Extremely generous, prioritises others over own resources | Highly materialistic, accumulates at others' expense |
| Pride | Deeply humble, deflects credit, minimises achievements | Ego-driven, seeks status and validation constantly |
| Lust | Very restrained, deliberate, avoids spontaneity | Highly impulsive, novelty-seeking, acts on urges |
| Wrath | Extreme conflict avoidance, never expresses anger | Quick to anger, confrontational, expresses frustration |
| Gluttony | Highly moderate, practices strict self-control | Strongly indulgent, struggles with restraint |
| Envy | Deeply content, never compares self to others | Constantly competitive, resentful of others' success |
| Sloth | Extremely proactive, takes initiative constantly | Avoidant, passive, procrastinates, takes easy path |

---

### 3.5 Multi-Observer Consensus Framework

**Research Basis:**

Huang et al. (2025, EMNLP Findings) demonstrated that multi-observer ratings outperform single-perspective assessment. Observer-report ratings align better with human judgments than self-reports; 5-7 observers provide a "wisdom of crowd" effect.

**Implementation:**

For responses flagged as ambiguous or for high-bias-risk traits (Wrath, Envy, Pride), Harmonia runs three parallel evaluations:

```
Observer 1 (Neutral Evaluator):

"You are an objective psychological assessor. Rate based only on

what is explicitly stated or clearly implied."


Observer 2 (Empathetic Therapist):

"You are a warm, understanding therapist. Consider what this person

might be feeling beneath their words."


Observer 3 (Skeptical Critic):

"You are analytically skeptical. Look for inconsistencies between

what the person claims and what their language reveals."
```

**Aggregation Logic:**

python

```
def aggregate_observers(scores: list[float]) -> dict:

    mean_score = statistics.mean(scores)
    std_dev = statistics.stdev(scores) if len(scores) > 1 else 0

    if std_dev < 0.3:
        # High agreement → authentic signal
        confidence_multiplier = 1.0
        flag = None
    elif std_dev < 0.5:
        # Moderate agreement → acceptable
        confidence_multiplier = 0.85
        flag = None
    else:
        # High disagreement → inconsistency detected
        confidence_multiplier = 0.6
        flag = "observer_disagreement"

    return {
        "score": round(mean_score, 2),
        "confidence_multiplier": confidence_multiplier,
        "std_dev": round(std_dev, 2),
        "flag": flag
    }
```

**When Observers Disagree:**

High standard deviation (>0.5) suggests defensive responding, genuinely ambiguous language, or cultural expression differences. These responses receive lower confidence weights in profile aggregation.

**API Call Optimisation:**

Full multi-observer (6 questions × 7 sins × 3 observers = 126 calls) is expensive. Recommended approach:

1. **Single observer for initial pass** — 42 calls total
2. **Multi-observer escalation** — Only for low-confidence or high-bias-risk traits
3. **Typical result** — ~50-60 calls per user

---

### 3.6 Social Desirability Bias Detection

**The Problem:**

Salecha et al. (2024, PNAS Nexus) discovered that LLMs identify personality evaluation contexts with 90% accuracy after just 5 questions. Once detected, scores systematically skew:

| Trait | Direction | Magnitude |
|-------|-----------|-----------|
| Extraversion | ↑ | +0.8 SD |

| | | |
|---|---|---|
| Conscientiousness | ↑ | +0.7 SD |
| Agreeableness | ↑ | +0.6 SD |
| Openness | ↑ | +0.5 SD |
| Neuroticism | ↓ | -1.1 SD |

GPT-4 shifted responses by 1.2 human standard deviations. This persists after question randomisation, paraphrasing, and temperature variation.

**Mitigation Strategy 1: Prompt-Level Reverse Coding**

Since Harmonia's 6 questions are fixed and not reverse-coded, the prompt handles this internally:

```
Standard interpretation: High positive emotion → Low Wrath

Reverse check: If response contains anger words but claims calm demeanor

          → Flag discrepancy, reduce confidence
```

This is less effective than true reverse-coded items (~50% bias reduction with proper design) but necessary given the fixed question set.

**Mitigation Strategy 2: Discrepancy Flagging**

Detect mismatches between claimed traits and linguistic behaviour:

python

```python
def detect_discrepancy(response_text: str, scores: dict) -> list[str]:

    flags = []

    text_lower = response_text.lower()


    # Wrath discrepancy
    anger_words = ["angry", "furious", "annoyed", "frustrated", "hate", "pissed"]

    anger_count = sum(1 for w in anger_words if w in text_lower)


    if scores["wrath"]["score"] < -2 and anger_count >= 2:

        flags.append("wrath_discrepancy: claims low anger but uses anger language")


    # Pride discrepancy

    self_promotion = ["best", "amazing", "incredible", "talented", "successful"]

    promo_count = sum(1 for w in self_promotion if w in text_lower)


    if scores["pride"]["score"] < -2 and promo_count >= 2:

        flags.append("pride_discrepancy: claims humility but self-promotes")


    return flags
```

**Mitigation Strategy 3: Cross-Response Consistency**

With only 6 questions, cross-response consistency is the primary bias detection mechanism:

python

```
def check_cross_response_consistency(all_question_results: list[dict]) -> dict:

    flags = []


    for sin in SINS:

        scores = [q["sins"][sin]["score"] for q in all_question_results]

        max_delta = max(scores) - min(scores)
        # With 6 questions, expect variation but not wild swings
        if max_delta > 5:  # e.g., -3 in Q1, +3 in Q4
            flags.append({
                "sin": sin,
                "flag": "inconsistent_across_responses",
                "max_delta": max_delta,
                "confidence_penalty": 0.25
            })

    return {"consistency_flags": flags}
```

---

## 3.7 LIWC Integration

**The Research Reality:**

Van Mil et al.'s meta-analysis (n=85,724 across 31 studies) found LIWC markers explain only 5.1% of Big Five variance. Effect sizes range |ρ| = 0.08 to 0.14. No single marker is diagnostic.

However, LIWC can serve as a **grounding signal** when combined with LLM interpretation.

**Most Predictive Markers:**

| Big Five Trait | Positive Markers | Negative Markers |
|---|---|---|
| Extraversion | 1st person plural, social words, positive emotion | Sadness, negative emotion |
| Agreeableness | Positive emotion, family, social words | Money, swearing, anger |

| | | |
|---|---|---|
| Conscientiousness | Future tense, work/achieve words, numbers | Filler words, negation |
| Neuroticism | Negative emotion, anxiety, health words | Positive emotion, certainty |
| Openness | Complex cognition, insight, tentative language | Certainty markers, past tense |

**Lightweight Extraction:**

python

```python
def extract_liwc_signals(text: str) -> dict:
    words = text.lower().split()
    word_count = len(words) or 1

    first_singular = ["i", "me", "my", "mine", "myself"]
    first_plural = ["we", "us", "our", "ours", "ourselves"]
    negative_emotion = ["angry", "sad", "afraid", "worried", "hate", "annoyed"]
    positive_emotion = ["happy", "love", "great", "excited", "wonderful"]
    certainty = ["always", "never", "definitely", "certainly", "absolutely"]
    hedging = ["maybe", "perhaps", "probably", "might", "sometimes"]
    future = ["will", "going to", "plan", "want to", "hope to"]

    return {
        "first_person_singular": sum(1 for w in words if w in first_singular) / word_count,
        "first_person_plural": sum(1 for w in words if w in first_plural) / word_count,
        "negative_emotion": sum(1 for w in words if w in negative_emotion) / word_count,
        "positive_emotion": sum(1 for w in words if w in positive_emotion) / word_count,
        "certainty": sum(1 for w in words if w in certainty) / word_count,
        "hedging": sum(1 for w in words if w in hedging) / word_count,
        "future_orientation": sum(1 for w in words if w in future) / word_count,
        "word_count": len(words)
    }
```

**Prompt Integration (Optional):**

```
Linguistic markers detected:
- First-person singular: 8% (moderate self-focus)
- Negative emotion words: 2% (low)
- Future-oriented language: 5% (moderate)

Combined with content analysis, score Sloth (-5 to +5): __
```

**Caveat:** LIWC alone is insufficient. Value comes from combining markers with semantic interpretation—but don't over-weight these signals given the weak effect sizes.

---

## 3.8 Generation Configuration

python

```
generation_config = {
    'thinking_level': 'low',
    'max_output_tokens': 1024,
    'response_mime_type': 'application/json',
    'response_json_schema': SINGLE_TRAIT_SCHEMA
}
```

**Parameter Rationale:**

| Parameter | Value | Reason |
|---|---|---|
| `thinking_level` | `'low'` | Trait parsing is structured extraction, not complex reasoning. Research shows deeper reasoning degrades personality inference. |
| `max_output_tokens` | `1024` | Single-trait output is small; leaves headroom for evidence quotes |
| `response_mime_type` | `'application/json'` | Guarantees valid JSON, eliminates parsing failures |
| `response_json_schema` | Schema object | Native schema enforcement for type safety |
| Temperature | *omitted* | Gemini 3 optimised for 1.0; low values cause looping |

**Single-Trait Schema:**

python

```
SINGLE_TRAIT_SCHEMA = {
    "type": "object",
    "properties": {
        "score": {"type": "number", "minimum": -5, "maximum": 5},

        "confidence": {"type": "number", "minimum": 0, "maximum": 1},

        "evidence": {"type": "string"}

    },

    "required": ["score", "confidence", "evidence"]

}
```

---

### 3.9 Safety Settings

python

```
safety_settings = [

    {"category": "HARM_CATEGORY_HARASSMENT", "threshold": "BLOCK_NONE"},

    {"category": "HARM_CATEGORY_HATE_SPEECH", "threshold": "BLOCK_NONE"},

    {"category": "HARM_CATEGORY_SEXUALLY_EXPLICIT", "threshold": "BLOCK_NONE"},

    {"category": "HARM_CATEGORY_DANGEROUS_CONTENT", "threshold":
"BLOCK_NONE"},

]
```

**Rationale:**

User responses contain authentic emotional content—descriptions of anger, jealousy, conflict, indulgence. A user writing "I'd probably lose my temper" provides valuable Wrath signal. Default safety filters would block legitimate personality assessment.

This is appropriate because Harmonia analyses user-provided self-descriptions rather than generating harmful content for distribution. Non-configurable protections (CSAM, PII) remain active.

---

### 3.10 Error Handling and Fallback

**Fallback Chain Execution:**

python

```python
async def parse_with_fallback(prompt: str) -> dict:
    for model_name in MODEL_CHAIN:
        try:
            model = genai.GenerativeModel(model_name)
            response = model.generate_content(
                prompt,
                generation_config=GENERATION_CONFIG,
                safety_settings=SAFETY_SETTINGS
            )
            return json.loads(response.text)

        except google.api_core.exceptions.ResourceExhausted:
            # Rate limit - try next model
            continue

        except google.api_core.exceptions.DeadlineExceeded:
            # Timeout - retry once, then fallback
            try:
                response = model.generate_content(prompt, ...)
                return json.loads(response.text)
            except:
                continue
        except json.JSONDecodeError:
            # Malformed response - try next model
            continue

        except Exception as e:
            logging.error(f"Unexpected error with {model_name}: {e}")
            continue

    # All models failed - return neutral with low confidence
    return {"score": 0, "confidence": 0.3, "evidence": ""}
```

**Safe Text Extraction:**

python

```
def safe_extract_text(response) -> str:
    """Handle various response structures."""
    try:
        return response.text
    except:
        pass

    try:
        return response.candidates[0].content.parts[0].text
    except:
        pass
    return ""
```

---

## 3.11 Expected Accuracy and Limitations

**Honest Assessment for 6 Fixed Questions:**

| Metric | Harmonia Expected | Research Benchmark |
|---|---|---|
| Pearson r vs. ground truth | ~0.20-0.25 | ~0.25-0.30 (3-5 questions, 150+ words) |
| Mean Absolute Error (1-5 scale) | ~0.9-1.1 | ~0.7-0.9 |
| Cronbach α (reliability) | ~0.50-0.60 | ~0.55-0.65 |

**Why Slightly Below Benchmark:**

1. **Fixed questions** — No randomisation means less trait-space coverage
2. **Variable word count** — Some responses may be 25 words, below 150+ threshold
3. **No reverse-coded items** — Social desirability bias harder to detect
4. **Novel framework** — Seven Deadly Sins has no validation data

**Practical Meaning:**

Expect scores to be off by ~1 point on the 10-point scale (-5 to +5). Enough for rough similarity matching, not enough for confident assessment.

**Critical Research Gaps:**

1. **No validation on dating scenarios** — All research uses essays, social media, clinical interviews
2. **No Seven Deadly Sins research** — Most work uses Big Five or MBTI
3. **No defence mechanism detection** — No peer-reviewed work on LLMs detecting projection, denial, rationalisation

**Appropriate Framing:**

| ✅ **Acceptable Claims** | ❌ **Overclaims** |
|---|---|
| "Personality indicator" | "Psychological assessment" |
| "Communication style matcher" | "Compatibility predictor" |
| "Perceived similarity signal" | "Scientific personality analysis" |

---

### 3.12 Complete Implementation

python

```
import google.generativeai as genai
import asyncio
import statistics
import json
import logging

class GeminiService:
    def __init__(self, api_key: str):
        genai.configure(api_key=api_key)

        self.model_chain = [
            'gemini-3-pro-preview',
            'gemini-3-flash-preview',
            'gemini-2.5-flash'
        ]

        self.sins = ["greed", "pride", "lust", "wrath", "gluttony", "envy", "sloth"]

        self.safety_settings = [
            {"category": "HARM_CATEGORY_HARASSMENT", "threshold": "BLOCK_NONE"},
            {"category": "HARM_CATEGORY_HATE_SPEECH", "threshold": "BLOCK_NONE"},
            {"category": "HARM_CATEGORY_SEXUALLY_EXPLICIT", "threshold":
"BLOCK_NONE"},
            {"category": "HARM_CATEGORY_DANGEROUS_CONTENT", "threshold":
"BLOCK_NONE"},
        ]

        self.generation_config = {
            'thinking_level': 'low',
            'max_output_tokens': 1024,
            'response_mime_type': 'application/json'
        }

        self.trait_anchors = {
            "greed": {
                "low": "Extremely generous, prioritises others over own resources",
                "high": "Highly materialistic, accumulates at others' expense"
```

```json
},
"pride": {
    "low": "Deeply humble, deflects credit, minimises achievements",
    "high": "Ego-driven, seeks status and validation constantly"
},
"lust": {
    "low": "Very restrained, deliberate, avoids spontaneity",
    "high": "Highly impulsive, novelty-seeking, acts on urges"
},
"wrath": {
    "low": "Extreme conflict avoidance, never expresses anger",
    "high": "Quick to anger, confrontational, expresses frustration"
},
"gluttony": {
    "low": "Highly moderate, practices strict self-control",
    "high": "Strongly indulgent, struggles with restraint"
},
"envy": {
    "low": "Deeply content, never compares self to others",
    "high": "Constantly competitive, resentful of others' success"
},
"sloth": {
    "low": "Extremely proactive, takes initiative constantly",
    "high": "Avoidant, passive, procrastinates, takes easy path"
```

```
        }
    }

async def parse_single_response(self, question: str, answer: str) -> dict:
    """Parse one question-response pair into seven sin scores."""

    # Run trait-specific parsing in parallel
    tasks = [
        self._parse_single_trait(question, answer, sin)
        for sin in self.sins
    ]

    trait_results = await asyncio.gather(*tasks)

    sins = {sin: result for sin, result in zip(self.sins, trait_results)}

    # Extract LIWC signals
    liwc = self._extract_liwc_signals(answer)

    # Detect discrepancies
    discrepancies = self._detect_discrepancies(answer, sins)

    return {
        "sins": sins,
        "liwc_signals": liwc,
        "discrepancies": discrepancies,
        "word_count": len(answer.split())
    }

async def _parse_single_trait(self, question: str, answer: str, sin: str) -> dict:
    """Parse single trait with fallback chain."""

    prompt = self._build_trait_prompt(question, answer, sin)

    for model_name in self.model_chain:
        try:
            model = genai.GenerativeModel(model_name)
            response = model.generate_content(
                prompt,
                generation_config=self.generation_config,
                safety_settings=self.safety_settings
            )
            return json.loads(response.text)
        except Exception as e:
            logging.warning(f"{model_name} failed for {sin}: {e}")
            continue

    return {"score": 0, "confidence": 0.3, "evidence": ""}
def _build_trait_prompt(self, question: str, answer: str, sin: str) -> str:
    anchors = self.trait_anchors[sin]

    return f"""Analyze this response for {sin.upper()} signals only.

{sin.capitalize()} Scale (-5 to +5):
-5 = {anchors['low']}
0 = Neutral: No clear signal
+5 = {anchors['high']}

Question: "{question}"
Response: "{answer}"

Return ONLY valid JSON:
{{"score": <-5 to +5>, "confidence": <0.0 to 1.0>, "evidence": "<direct quote or empty>"}}"""

def _extract_liwc_signals(self, text: str) -> dict:
    words = text.lower().split()
    wc = len(words) or 1

    return {
        "first_person_singular": sum(1 for w in words if w in ["i","me","my","mine"]) / wc,
        "first_person_plural": sum(1 for w in words if w in ["we","us","our","ours"]) / wc,
        "negative_emotion": sum(1 for w in words if w in
["angry","sad","hate","annoyed","frustrated"]) / wc,
        "positive_emotion": sum(1 for w in words if w in ["happy","love","great","excited"]) / wc,
        "certainty": sum(1 for w in words if w in ["always","never","definitely"]) / wc,
        "hedging": sum(1 for w in words if w in ["maybe","perhaps","probably","might"]) / wc,
    }

def _detect_discrepancies(self, text: str, sins: dict) -> list[str]:
    flags = []
    text_lower = text.lower()

    # Wrath discrepancy
    anger_words = ["angry", "furious", "annoyed", "frustrated", "hate"]
    if sins["wrath"]["score"] < -2 and sum(1 for w in anger_words if w in text_lower) >= 2:
        flags.append("wrath_discrepancy")

    # Pride discrepancy
    humble_claim = sins["pride"]["score"] < -2
    self_promo = ["best", "amazing", "incredible", "talented"]
    if humble_claim and sum(1 for w in self_promo if w in text_lower) >= 2:
        flags.append("pride_discrepancy")

    return flags

async def parse_all_responses(self, responses: list[dict]) -> dict:
    """Parse all 6 question-response pairs and aggregate."""

    # Parse each response
    tasks = [
        self.parse_single_response(r["question"], r["answer"])
        for r in responses
    ]

    all_results = await asyncio.gather(*tasks)

    # Aggregate across questions
    aggregated = self._aggregate_profile(all_results)

    # Check cross-response consistency
    consistency = self._check_consistency(all_results)

    return {
        "per_question": all_results,
        "aggregated_profile": aggregated,
        "consistency_flags": consistency,
        "total_words": sum(r["word_count"] for r in all_results)
    }

def _aggregate_profile(self, all_results: list[dict]) -> dict:
```

```
"""Confidence-weighted aggregation across 6 questions."""

aggregated = {}

for sin in self.sins:
    scores = []
    weights = []
    all_evidence = []

    for result in all_results:
        s = result["sins"][sin]
        scores.append(s["score"])
        weights.append(s["confidence"])
        if s["evidence"]:
            all_evidence.append(s["evidence"])

    # Weighted average
    total_weight = sum(weights) or 1
    weighted_score = sum(s * w for s, w in zip(scores, weights)) / total_weight

    aggregated[sin] = {
        "score": round(weighted_score, 2),
        "confidence": round(statistics.mean(weights), 2),
        "evidence_samples": all_evidence[:2]  # Top 2 quotes
        }

        return aggregated

    def _check_consistency(self, all_results: list[dict]) -> list[dict]:
        """Flag traits with high variance across questions."""

        flags = []

        for sin in self.sins:
            scores = [r["sins"][sin]["score"] for r in all_results]
            max_delta = max(scores) - min(scores)

            if max_delta > 5:
                flags.append({
                    "sin": sin,
                    "max_delta": max_delta,
                    "flag": "high_variance_across_responses"
                })

        return flags
```

---

## 3.13 Summary

**What Harmonia's Gemini Integration Achieves:**

| Capability | Implementation | Expected Accuracy |
|---|---|---|
| Trait scoring | 7 trait-specific prompts per question | MAE ~0.9-1.1 |
| Bias detection | Discrepancy flags + cross-response consistency | Catches ~30% of performative responses |
| Confidence calibration | Per-trait confidence + aggregation weighting | Flags low-signal responses appropriately |
| Reliability | 6-question aggregation | α ~0.50-0.60 (borderline) |

**What It Doesn't Achieve:**

- ❌ Diagnostic-level accuracy (would require ground-truth fine-tuning)
- ❌ Defence mechanism detection (no validated research exists)
- ❌ Relationship outcome prediction (Joel et al. 2017 established this is impossible from pre-meeting data)

**The Honest Value Proposition:**

Harmonia's personality parsing captures **self-presentation style** and **communication patterns**. Two users who frame situations similarly, use similar emotional vocabulary, and express similar values when responding to hypotheticals may genuinely enjoy each other's company.

This is not deep psychological compatibility. It's **perceived similarity**—and research shows perceived similarity correlates with initial attraction even when "actual" compatibility doesn't.

The system works not because it accurately measures personality, but because it surfaces genuine overlap in how people present themselves. For a dating app's first filter, that may be exactly what's needed.

# Section 4: Scoring Mechanics

## 4.1 The Bipolar Scale: -5 to +5

Harmonia uses an 11-point bipolar scale ranging from -5 (extreme virtue) through 0 (neutral) to +5 (extreme vice). This design captures the full spectrum of each personality dimension rather than treating sins as purely negative traits.

**Why Bipolar Rather Than Unipolar:**

A unipolar scale (0-10) implies that "0 Greed" means absence of the trait. But psychologically, the opposite of greed isn't "no greed"—it's generosity. Someone who scores -4 on Greed isn't merely "not greedy"; they actively prioritise others' needs over their own resources. This distinction matters for compatibility matching: two people who both score -3 on Wrath share a conflict-avoidant style, which is meaningful information.

Research supports this approach. Woods & Hampson (2005) demonstrated that bipolar scales with anchored endpoints show better construct validity for personality dimensions where both poles are psychologically meaningful. The semantic differential tradition (Osgood) established that bipolar adjective pairs capture attitudinal nuance that unipolar scales miss.

**Why 11 Points (-5 to +5):**

Psychometric research indicates:

| Scale Length | Finding |
|---|---|
| 5-point | Acceptable but 23% lower response variance than 7-point |
| 7-point | Optimal balance of reliability, validity, and usability |
| 11-point | Higher reliability than 7-point; superior criterion validity; preferred by educated respondents |

For Harmonia's use case—parsing free-text responses where Gemini must place nuanced linguistic signals on a continuum—11 points provides the granularity needed to distinguish between, say, "mildly generous" (-2) and "notably generous" (-3). The marginal cognitive burden is irrelevant since users never see or interact with the scale directly; it's purely internal.

**Score Interpretation:**

| Score | Interpretation | Example (Wrath) |
|---|---|---|
| -5 | Extreme virtue | "I would never confront anyone, even if they wronged me badly" |
| -3 | Strong virtue | "I always try to understand their perspective before reacting" |
| -1 | Slight virtue | "I prefer to let things go rather than argue" |
| 0 | Neutral / No signal | Topic not addressed, or conflicting signals cancel out |
| +1 | Slight vice | "That would probably annoy me" |
| +3 | Strong vice | "I'd definitely say something—I can't let that slide" |
| +5 | Extreme vice | "I'd lose my temper and tell them exactly what I think" |

**The Critical Meaning of Zero:**

Research reveals a fundamental ambiguity in bipolar midpoint ratings. Anvari & Lakens (2023) demonstrated that midpoint selections conflate two psychologically distinct states:

1. **True neutrality** — Neither pole applies; the person is genuinely average on this dimension
2. **Ambivalence** — Both poles apply in different contexts; the person exhibits contradictory tendencies
3. **No signal** — Insufficient information to determine position

For Harmonia's LLM-parsed responses, a score of 0 should be interpreted as **"no strong signal detected"** rather than "average personality." When Gemini cannot extract a clear directional signal from the response text, assigning 0 with low confidence accurately represents epistemic uncertainty rather than making a substantive claim about the person's trait level.

This interpretation has practical implications: a 0 score should widen confidence intervals and receive lower weight in similarity calculations, not be treated as definitive evidence of trait neutrality.

---

## 4.2 Sin Definitions and Behavioural Mapping

Each of the Seven Deadly Sins maps to established psychological constructs while providing a more memorable and marketable framework. The definitions below specify what Gemini should detect in user responses.

---

**GREED**

**Core Definition:** Orientation toward resource accumulation, materialism, and self-interest in resource allocation.

| Pole | Description |
|---|---|
| -5 (Generosity) | Prioritises others' needs; gives freely without expectation; uncomfortable accumulating while others lack |
| 0 (Neutral) | No clear signal about resource orientation |
| +5 (Greed) | Prioritises personal accumulation; calculates advantage in transactions; reluctant to share resources |

**Big Five Mapping:** Agreeableness (inverse) — Low agreeableness correlates with competitive, self-interested resource behaviour.

**Linguistic Markers:**

- Vice indicators: "my share," "what I'm owed," "fair split," cost-benefit language, calculation references
- Virtue indicators: "happy to cover," "doesn't matter who pays," "I'd rather give," resource-sharing language

**Behavioural Examples:**

| Score | Example Response |
|---|---|
| -4 | "I'd insist on paying for everyone—it's just money, and I like treating people" |
| -1 | "I don't really keep track of who owes what" |
| +2 | "I'd want to make sure everyone pays their fair share" |
| +4 | "I always calculate exactly what I owe, down to the penny" |

---

**PRIDE**

**Core Definition:** Orientation toward status, external validation, ego protection, and concern with how one is perceived.

| Pole | Description |
|---|---|
| -5 (Humility) | Deflects credit; minimises achievements; uncomfortable with praise; focuses on others' contributions |
| 0 (Neutral) | No clear signal about status orientation |
| +5 (Pride) | Seeks recognition; concerned with reputation; emphasises own accomplishments; sensitive to perceived slights |

**Big Five Mapping:** Extraversion (status-seeking facet), Neuroticism inverse (ego fragility).

**Linguistic Markers:**

- Vice indicators: "people would think," "my reputation," "I deserve," achievement emphasis, comparison to others' status
- Virtue indicators: "it wasn't just me," "anyone could have," deflection of praise, focus on team/others

**Behavioural Examples:**

| Score | Example Response |
|---|---|
| -3 | "I'd rather not take credit—the whole team made it happen" |
| 0 | Response discusses the situation without self-reference |
| +3 | "I'd make sure people knew about my contribution" |
| +5 | "I worked harder than anyone else, and they need to recognise that" |

---

**LUST**

**Core Definition:** Orientation toward impulsivity, novelty-seeking, spontaneity, and immediate gratification versus deliberation and restraint.

| Pole | Description |
|---|---|
| -5 (Restraint) | Highly deliberate; plans carefully; avoids spontaneity; uncomfortable with uncertainty |
| 0 (Neutral) | No clear signal about impulsivity orientation |

| | |
|---|---|
| +5 (Lust) | Acts on urges; seeks novelty; prioritises excitement; makes quick decisions without extensive deliberation |

**Big Five Mapping:** Openness to Experience (novelty-seeking facet), Conscientiousness inverse (impulsivity).

**Linguistic Markers:**

- Vice indicators: "spontaneous," "why not," "live in the moment," "try anything once," excitement-seeking language
- Virtue indicators: "I'd think it through," "need to plan," "careful consideration," "not worth the risk"

**Behavioural Examples:**

| Score | Example Response |
|---|---|
| -4 | "I'd need to research all the options and make a spreadsheet before deciding" |
| -1 | "I'd probably sleep on it before committing" |
| +2 | "Sounds fun—I'd probably just go for it" |
| +5 | "Life's too short to overthink—I'd say yes immediately" |

---

**WRATH**

**Core Definition:** Orientation toward conflict, anger expression, confrontation versus conflict avoidance and emotional regulation.

| Pole | Description |
|---|---|
| -5 (Calm) | Extreme conflict avoidance; never expresses anger; prioritises harmony over addressing issues |
| 0 (Neutral) | No clear signal about conflict orientation |
| +5 (Wrath) | Quick to anger; confrontational; expresses frustration readily; addresses conflicts directly and forcefully |

**Big Five Mapping:** Agreeableness inverse (antagonism), Neuroticism (anger/hostility facet).

**Linguistic Markers:**

- Vice indicators: "that would piss me off," "I'd tell them," "confront," "can't let that go," anger words
- Virtue indicators: "avoid drama," "let it go," "not worth fighting over," "keep the peace," harmony language

**Behavioural Examples:**

| Score | Example Response |
|---|---|
| -4 | "I'd never say anything—it's not worth the conflict" |
| -1 | "I'd probably just let it slide this time" |
| +2 | "I'd definitely bring it up with them" |
| +5 | "I'd lose my temper and tell them exactly what I think of their behaviour" |

**Note on Wrath Weighting:** Research on relationship satisfaction consistently identifies conflict style as highly predictive. Harmonia weights Wrath at 1.5× in similarity calculations.

---

## GLUTTONY

**Core Definition:** Orientation toward excess, indulgence, and lack of moderation versus restraint and self-control.

| Pole | Description |
|---|---|
| -5 (Moderation) | Practices strict self-control; uncomfortable with excess; prefers restraint in all things |
| 0 (Neutral) | No clear signal about moderation orientation |
| +5 (Gluttony) | Indulgent; struggles with restraint; "more is more" mentality; justifies excess |

**Big Five Mapping:** Conscientiousness inverse (self-discipline facet).

**Linguistic Markers:**

- Vice indicators: "treat myself," "I deserve," "just one more," "why not indulge," excess language

- Virtue indicators: "that's enough," "moderation," "don't need more," restraint language

**Behavioural Examples:**

| Score | Example Response |
|---|---|
| -3 | "I'd stop after one—no need to overdo it" |
| 0 | Response doesn't address consumption/indulgence |
| +3 | "If it's good, why not have more?" |
| +5 | "I'd keep going until it was all gone—you only live once" |

---

**ENVY**

**Core Definition:** Orientation toward social comparison, competition, and resentment of others' success versus contentment and support for others.

| Pole | Description |
|---|---|
| -5 (Contentment) | Never compares self to others; genuinely celebrates others' success; satisfied with own situation |
| 0 (Neutral) | No clear signal about comparison orientation |
| +5 (Envy) | Constantly compares; competitive; resentful of others' advantages; "keeping up" mentality |

**Big Five Mapping:** Neuroticism (resentment), Agreeableness inverse (competitiveness).

**Linguistic Markers:**

- Vice indicators: "they have," "why do they get," "not fair that," comparison language, competitive framing
- Virtue indicators: "good for them," "happy they succeeded," "doesn't affect me," contentment language

**Behavioural Examples:**

| Score | Example Response |
|---|---|
| -4 | "I'm genuinely happy for them—their success doesn't diminish mine" |

| | |
|---|---|
| -1 | "Good for them, I guess" |
| +2 | "I'd wonder why they got that and I didn't" |
| +5 | "It's not fair—I work just as hard and never get those opportunities" |

---

**SLOTH**

**Core Definition:** Orientation toward avoidance, passivity, procrastination, and path of least resistance versus proactivity and initiative.

| Pole | Description |
|---|---|
| -5 (Proactivity) | Takes initiative constantly; uncomfortable with inaction; energetic approach to challenges |
| 0 (Neutral) | No clear signal about effort orientation |
| +5 (Sloth) | Avoids effort; procrastinates; takes easy path; passive in face of challenges |

**Big Five Mapping:** Conscientiousness inverse (achievement-striving), Extraversion inverse (activity level).

**Linguistic Markers:**

- Vice indicators: "too much effort," "deal with it later," "someone else can," "path of least resistance"
- Virtue indicators: "I'd get started right away," "take action," "handle it myself," initiative language

**Behavioural Examples:**

| Score | Example Response |
|---|---|
| -4 | "I'd tackle it immediately—no point in waiting" |
| -1 | "I'd probably get around to it soon" |
| +2 | "I'd put it off until I absolutely had to" |
| +5 | "Honestly, I'd just hope it resolved itself" |

**Note on Sloth Weighting:** Motivation alignment matters for long-term compatibility. Harmonia weights Sloth at 1.3× in similarity calculations.

---

## 4.3 Confidence Scoring

Confidence represents **epistemic certainty about the score**—how sure Gemini is that the assigned value accurately reflects the user's personality signal. It is NOT a measure of trait strength. A score of +5 with 0.60 confidence means "if this person has high Wrath, it's extreme—but I'm not certain they have high Wrath."

**Research-Backed Confidence Thresholds:**

LLM calibration research (2024-2025) establishes that confidence scores correlate with accuracy in predictable bands. Below 0.70, accuracy approaches zero; above 0.95, accuracy is substantially higher.

| Confidence Range | Interpretation | Evidence Type | Accuracy Expectation |
|---|---|---|---|
| 0.95-1.00 | **High confidence** | Explicit statement | ~95% accuracy |
| 0.70-0.94 | **Medium confidence** | Multiple converging cues | Variable, usable |
| 0.50-0.69 | **Low confidence** | Weak or single cue | Near-chance |
| 0.00-0.49 | **Very low confidence** | Speculative | Exclude from decisions |

**What Each Level Means in Practice:**

**High Confidence (≥0.95):**

- User explicitly stated the trait: "I always lose my temper"
- Clear, unambiguous language directly addressing the dimension
- Example: Response to conflict scenario includes "I'd be furious and tell them off"
- **Action:** Trust this score; weight fully in aggregation

**Medium Confidence (0.70-0.94):**

- Multiple linguistic cues converge on the same direction
- Implied but not explicitly stated
- Example: User describes scenario resolution that *implies* conflict avoidance without saying "I avoid conflict"

- **Action:** Include in aggregation; weight by confidence value

**Low Confidence (0.50-0.69):**

- Single weak cue or ambiguous language
- Could reasonably be interpreted multiple ways
- Example: "I'd probably just deal with it" (could be proactive or avoidant depending on tone)
- **Action:** Include with reduced weight; flag if this is the only signal for this trait

**Very Low Confidence (0.00-0.49):**

- Topic not addressed in response
- Contradictory signals that cancel out
- Insufficient text to make any inference
- **Action:** Assign score of 0; effectively exclude from similarity calculations via low weight

**The Calibration Problem:**

Research shows LLMs are systematically overconfident—self-reported confidence exceeds actual accuracy. Gemini's confidence scores should be treated as **relative rankings** (higher = more certain) rather than **calibrated probabilities** (0.80 = 80% accurate).

To mitigate this:

1. Empirically validate confidence thresholds against ground-truth personality data during beta
2. Calculate Expected Calibration Error (ECE) for each trait
3. Adjust thresholds if ECE exceeds 0.15

**Confidence Prompt Guidance:**

The prompt instructs Gemini to assign confidence based on linguistic explicitness:

```
Confidence Guidelines:
- 0.95-1.00: User explicitly states this trait ("I always...", "I never...")
- 0.80-0.94: Strong implication from multiple cues in the response
- 0.60-0.79: Moderate inference from context and tone
- 0.40-0.59: Weak inference; could be interpreted otherwise

- 0.00-0.39: No signal; trait not addressed in response
```

---

## 4.4 Evidence Extraction

For each sin score, Gemini extracts a direct quote from the user's response that supports the assigned value. Evidence serves three purposes:

1. **Interpretability** — Humans reviewing the system can understand why a score was assigned
2. **Debugging** — When scores seem wrong, evidence reveals parsing errors
3. **Auditability** — If users question their matches, evidence provides transparent justification

**What Makes Good Evidence:**

| Quality | Description | Example |
|---|---|---|
| **Direct** | Exact quote from response, not paraphrase | ✅ "I'd probably just let it slide" |
| **Relevant** | Clearly supports the specific sin being scored | ✅ For Wrath: "avoid the drama" |
| **Minimal** | Shortest quote that captures the signal | ✅ "not worth fighting over" vs. entire paragraph |
| **Strongest** | If multiple quotes apply, select the most diagnostic | ✅ Most explicit statement |

**Evidence Field Behaviour:**

| Scenario | Evidence Field | Confidence |
|---|---|---|
| Clear quote supports score | Direct quote | 0.70+ |
| Multiple relevant quotes | Select strongest single quote | 0.80+ |
| Implied but no quotable phrase | Empty string " " | 0.50-0.70 |
| No signal for this trait | Empty string " " | 0.00-0.50 |
| Contradictory quotes | Empty string " " + flag discrepancy | 0.40-0.60 |

**Handling Edge Cases:**

**No Clear Quote:** When the score is inferred from overall tone rather than specific language, Gemini returns an empty evidence string. This automatically correlates with lower confidence—if there's no quotable evidence, the inference is weaker.

**Multiple Relevant Quotes:** Select the single strongest quote. If a user says both "I'd probably let it go" and "not worth the argument," choose whichever is more diagnostically clear for the target sin.

**Contradictory Quotes:** When a response contains conflicting signals ("I usually avoid conflict, but this would really piss me off"), Gemini should:

1. Assign a score reflecting the net direction (or 0 if truly balanced)
2. Reduce confidence to reflect uncertainty
3. Leave evidence empty or select the quote matching the assigned direction
4. The discrepancy detection system (Section 3.6) flags this for review

**Evidence is Internal:**

Users never see evidence quotes. They exist purely for system validation and debugging. The user-facing experience shows only match percentages and positive overlap highlights, not the underlying scoring mechanics.

---

## 4.5 Aggregation Across Questions

Each user answers 6 questions. Each question generates 7 sin scores (one per sin). The aggregation system combines these 42 data points into a single 7-sin profile.

**Confidence-Weighted Mean Formula:**

Research supports confidence-weighted majority voting (CWMV) when confidence varies meaningfully across items:

```
Score_trait = Σ(Item_score_i × Confidence_i) / Σ(Confidence_i)
```

For example, aggregating Wrath scores across 6 questions:

| Question | Wrath Score | Confidence | Weighted Contribution |
|---|---|---|---|
| Q1 | -2 | 0.85 | -1.70 |
| Q2 | -3 | 0.90 | -2.70 |
| Q3 | 0 | 0.30 | 0.00 |

| | | | |
|---|---|---|---|
| Q4 | -1 | 0.65 | -0.65 |
| Q5 | -2 | 0.80 | -1.60 |
| Q6 | +1 | 0.45 | +0.45 |

Sum of weighted scores: -6.20
Sum of confidences: 3.95

Aggregated Wrath score: -6.20 / 3.95 = -1.57 ≈ -1.6

Notice how Q3 (no signal, low confidence) and Q6 (contradictory signal, low confidence) have minimal impact on the final score, while Q2 (strong signal, high confidence) dominates.

**When to Use CWMV vs. Simple Mean:**

Research indicates CWMV is superior when:

- Confidence calibration is acceptable (ECE ≤ 0.15)
- Confidence varies substantially across items (SD ≥ 0.10)

Until Harmonia validates Gemini's confidence calibration against ground truth, implement both:

python
```
def aggregate_trait(scores: list[dict]) -> dict:
    values = [s["score"] for s in scores]
    confidences = [s["confidence"] for s in scores]

    confidence_sd = statistics.stdev(confidences) if len(confidences) > 1 else 0

    # Use CWMV if confidence varies meaningfully
    if confidence_sd >= 0.10:
        weighted_sum = sum(v * c for v, c in zip(values, confidences))
        confidence_sum = sum(confidences)
        aggregated_score = weighted_sum / confidence_sum if confidence_sum > 0 else 0
        method = "cwmv"
    else:
        # Fall back to simple mean if confidence is uniform
        aggregated_score = statistics.mean(values)
        method = "mean"

    return {
        "score": round(aggregated_score, 2),
        "confidence": round(statistics.mean(confidences), 2),
        "method": method,
        "item_count": len(scores),
        "variance": round(statistics.variance(values), 2) if len(values) > 1 else 0
    }
```

**Handling High Variance (Conflicting Signals):**

When scores for the same trait vary substantially across questions, this could indicate:

1. **Genuine complexity** — Person behaves differently in different contexts
2. **Measurement error** — Questions elicited unreliable signals
3. **Response style** — User answered inconsistently

Research suggests flagging when within-person standard deviation exceeds 2.0 on a 7-point scale (equivalent to ~3.0 on our 11-point scale):

python
```python
def check_trait_consistency(scores: list[dict]) -> dict:
    values = [s["score"] for s in scores]
    sd = statistics.stdev(values) if len(values) > 1 else 0

    if sd > 3.0:
        return {
            "flag": "high_variance",
            "sd": round(sd, 2),
            "recommendation": "reduce_weight_in_similarity"
        }
    return {"flag": None}
```

High-variance traits receive reduced weight in similarity calculations—not because the person is unknowable, but because our confidence in the aggregated score is lower.

**Aggregated Confidence:**

The final confidence for each trait is the mean of item confidences, adjusted for variance:

python
```python
def calculate_aggregated_confidence(scores: list[dict]) -> float:
    confidences = [s["confidence"] for s in scores]
    base_confidence = statistics.mean(confidences)

    values = [s["score"] for s in scores]
    sd = statistics.stdev(values) if len(values) > 1 else 0

    # Penalize confidence if scores are inconsistent
    if sd > 2.0:
        variance_penalty = min(0.3, (sd - 2.0) * 0.1)
        return max(0.1, base_confidence - variance_penalty)

    return base_confidence
```

**Final Profile Structure:**

After aggregating all 6 questions, the user profile contains:

python
```
{
    "user_id": "abc123",
    "profile": {
        "greed": {"score": -1.2, "confidence": 0.72, "variance": 1.8},
        "pride": {"score": +0.5, "confidence": 0.58, "variance": 2.4},
        "lust": {"score": +2.8, "confidence": 0.81, "variance": 1.2},
        "wrath": {"score": -2.1, "confidence": 0.85, "variance": 0.9},
        "gluttony": {"score": +1.4, "confidence": 0.65, "variance": 2.1},
        "envy": {"score": -0.8, "confidence": 0.70, "variance": 1.5},
        "sloth": {"score": -1.5, "confidence": 0.77, "variance": 1.1}
    },
    "flags": ["pride_high_variance"],
    "total_words": 847,
    "questions_answered": 6
}
```

---

### 4.6 Trait Weights for Similarity Calculation

Not all sins contribute equally to perceived compatibility. Based on relationship research and Felix's framework, Harmonia applies differential weights:

| Sin | Weight | Rationale |
|-----|--------|-----------|
| Wrath | 1.5× | Conflict style is the strongest predictor of relationship satisfaction |
| Sloth | 1.3× | Motivation alignment matters for shared activities and life goals |
| Pride | 1.2× | Ego dynamics affect communication and mutual respect |
| Lust | 1.0× | Baseline weight for novelty-seeking and spontaneity |
| Greed | 0.9× | Resource attitudes matter but less than interpersonal dynamics |
| Gluttony | 0.8× | Lifestyle compatibility signal but lower relationship impact |
| Envy | 0.7× | Comparison orientation affects satisfaction but indirectly |

These weights apply during similarity calculation (Section 5), not during score aggregation. A person's Envy score is calculated the same way as their Wrath score; Envy simply contributes less to the final match percentage.

---

### 4.7 Reliability Standards

For a 6-item personality indicator used in low-stakes matching (not clinical diagnosis), psychometric research establishes:

| Metric | Minimum Acceptable | Preferred |
|--------|-------------------|-----------|
| Cronbach's α | ≥ 0.60 | 0.70-0.80 |
| Average inter-item correlation | ≥ 0.30 | 0.40-0.50 |
| Within-person SD | < 3.0 | < 2.0 |

**Expected Reliability by Trait:**

Based on Big Five research, some traits naturally achieve higher internal consistency:

| Trait Type | Expected α | Harmonia Equivalent |
|------------|------------|---------------------|
| Neuroticism-adjacent | 0.75-0.85 | Wrath, Envy |
| Conscientiousness-adjacent | 0.75-0.85 | Sloth, Gluttony |
| Agreeableness-adjacent | 0.65-0.75 | Greed |
| Openness-adjacent | 0.65-0.75 | Lust |
| Extraversion-adjacent | 0.70-0.80 | Pride |

If any trait consistently falls below α = 0.60 in beta testing, investigate:
1. Question coverage — Are all 6 questions capable of eliciting this trait?
2. Trait definition — Is the sin definition too broad or ambiguous?
3. Parsing accuracy — Is Gemini misinterpreting linguistic cues for this trait?

---

### 4.8 Summary

**Scoring Mechanics Overview:**

| Component | Specification |
|-----------|---------------|
| Scale | -5 to +5 bipolar (virtue to vice) |
| Midpoint (0) | "No strong signal" not "average" |

| | |
|---|---|
| Confidence | 0.00-1.00; thresholds at 0.70 and 0.95 |
| Evidence | Direct quote or empty string |
| Aggregation | Confidence-weighted mean (CWMV) |
| Variance flag | SD > 3.0 across questions |
| Minimum reliability | Cronbach's α ≥ 0.60 |

**The Interpretive Hierarchy:**

```
High confidence (≥0.95) + Consistent scores (SD < 2.0)
→ Strong signal; weight fully in similarity

Medium confidence (0.70-0.94) + Consistent scores
→ Usable signal; weight by confidence

Any confidence + High variance (SD > 3.0)
→ Weak signal; reduce weight, flag for review

Low confidence (<0.70) + Inconsistent

→ Minimal signal; contributes little to similarity
```

**What This Achieves:**

The scoring mechanics ensure that:

1. Strong, explicit signals dominate the personality profile
2. Weak or ambiguous signals are appropriately downweighted
3. Conflicting information is flagged rather than hidden
4. The system's uncertainty is transparent and quantified

This doesn't make the scores *accurate* in an absolute sense—Section 3 established that LLM-based personality inference has fundamental limitations. But it ensures that whatever signal exists is extracted and weighted appropriately, and that the system knows what it doesn't know.

## Section 5: Profile Aggregation

### 5.1 The Aggregation Pipeline

Profile aggregation transforms raw Gemini outputs into a single, quality-controlled personality profile. The pipeline processes 42 individual data points (6 questions × 7 sins) into 7 aggregated trait scores with associated confidence and quality metrics.

**Data Flow Overview:**

```
User Responses (6 questions, 25-150 words each)
            ↓
      GeminiService Parsing
            ↓
  42 Trait-Question Scores (7 sins × 6 questions)
  Each with: score (-5 to +5), confidence (0-1), evidence
            ↓
      Question-Level Quality Control
      - Word count validation
      - Response completeness check
      - Individual Q&A rejection criteria
            ↓
      Trait-Level Aggregation
      - CWMV or simple mean per sin
      - Variance calculation
      - Consistency flagging
            ↓
      Profile-Level Quality Control
      - Response style detection (ERS, MRS, patterns)
      - Composite quality score
      - Minimum viability check
            ↓
  Final Profile (7 aggregated sins + metadata)
```

**Why This Order Matters:**

Quality control operates at three levels, each catching different problems:

1. **Question-level** catches individual bad responses (too short, off-topic)
2. **Trait-level** catches inconsistent signals within a single sin
3. **Profile-level** catches systematic response styles affecting all traits

A response might pass question-level checks (sufficient words, on-topic) but contribute to a profile-level flag (all responses use extreme language, suggesting ERS). The pipeline must process all levels before determining profile viability.

---

## 5.2 Question-Level Quality Control

Before aggregation, each of the 6 question-response pairs undergoes validation. Problems at this level affect all 7 sin scores derived from that response.

**Validation Checks:**

| Check | Threshold | Action if Failed |
|---|---|---|
| Word count | < 25 words | Exclude Q&A from aggregation |
| Word count | 25-50 words | Include but reduce confidence by 0.2 |
| Word count | > 150 words | Include normally (no bonus) |
| Response completeness | Empty or "N/A" | Exclude Q&A from aggregation |
| Off-topic detection | Gemini returns all 0s with confidence < 0.3 | Exclude Q&A from aggregation |

**Minimum Questions Required:**

With 6 total questions, how many must pass validation for a viable profile?

| Questions Passed | Action |
|---|---|
| 6 of 6 | Proceed normally |
| 5 of 6 | Proceed with reduced profile confidence |
| 4 of 6 | Proceed with warning flag; recommend re-assessment |
| 3 or fewer | Reject profile; require user to answer more questions |

Research on ultra-short scales indicates that below 4 items per trait, reliability collapses entirely. Since each question contributes one data point per sin, 4 questions represents the absolute minimum for any statistical aggregation.

**Implementation:**

```
def validate_question_response(question: str, response: str, parsed: dict) -> dict:
    """
    Validate a single Q&A pair before including in aggregation.

    Returns:
        {
            "valid": bool,
            "confidence_penalty": float,  # 0.0 to 0.2
            "exclusion_reason": str | None
        }
    """
    word_count = len(response.split())

    # Check word count
    if word_count < 25:
        return {
            "valid": False,
            "confidence_penalty": 0.0,
            "exclusion_reason": "insufficient_words"
        }

    # Check for empty/placeholder responses
    placeholder_patterns = ["n/a", "na", "none", "skip", "no answer", "..."]
    if response.strip().lower() in placeholder_patterns:
        return {
            "valid": False,
            "confidence_penalty": 0.0,
            "exclusion_reason": "placeholder_response"
        }

    # Check if Gemini found any signal
    all_scores = [parsed["sins"][sin]["score"] for sin in parsed["sins"]]
    all_confidences = [parsed["sins"][sin]["confidence"] for sin in parsed["sins"]]

    if all(s == 0 for s in all_scores) and statistics.mean(all_confidences) < 0.3:
        return {
            "valid": False,
            "confidence_penalty": 0.0,
            "exclusion_reason": "no_signal_detected"
        }

    # Passed validation
    confidence_penalty = 0.2 if word_count < 50 else 0.0

    return {
        "valid": True,
        "confidence_penalty": confidence_penalty,
        "exclusion_reason": None
    }
```

---

## 5.3 Trait-Level Aggregation

After question-level validation, each sin has up to 6 data points (one per valid question). Aggregation combines these into a single score per sin.

**The Core Decision: CWMV vs. Simple Mean**

Section 4.5 established the confidence-weighted mean formula:

$Score\_trait = \Sigma(Item\_score\_i \times Confidence\_i) / \Sigma(Confidence\_i)$

Research indicates CWMV outperforms simple mean when:

- Confidence calibration is acceptable (ECE ≤ 0.15)
- Confidence varies meaningfully across items (SD ≥ 0.10)

Until Harmonia validates Gemini's confidence calibration empirically, the system uses a **conditional approach**:

```
def aggregate_single_trait(scores: list[dict]) -> dict:
    """
    Aggregate multiple question scores for one sin into a single score.

    Args:
        scores: List of {"score": float, "confidence": float, "evidence": str}

    Returns:
        {
            "score": float,
            "confidence": float,
            "variance": float,
            "item_count": int,
            "method": str,
            "high_variance_flag": bool
        }
    """
    if not scores:
        return {
            "score": 0.0,
            "confidence": 0.0,
            "variance": 0.0,
            "item_count": 0,
            "method": "none",
            "high_variance_flag": False
        }

    values = [s["score"] for s in scores]
    confidences = [s["confidence"] for s in scores]

    # Calculate variance
    variance = statistics.variance(values) if len(values) > 1 else 0.0

    # Determine aggregation method
    confidence_sd = statistics.stdev(confidences) if len(confidences) > 1 else 0.0

    if confidence_sd >= 0.10:
        # Use CWMV - confidence varies meaningfully
        weighted_sum = sum(v * c for v, c in zip(values, confidences))
        confidence_sum = sum(confidences)
        aggregated_score = weighted_sum / confidence_sum if confidence_sum > 0 else 0.0
        method = "cwmv"
    else:
        # Fall back to simple mean - confidence is uniform
        aggregated_score = statistics.mean(values)
        method = "mean"

    # Calculate aggregated confidence (penalized by variance)
    base_confidence = statistics.mean(confidences)

    # High variance threshold: SD > 3.0 on 11-point scale
    # (equivalent to SD > 2.0 on 7-point scale from research)
    score_sd = statistics.stdev(values) if len(values) > 1 else 0.0
    high_variance = score_sd > 3.0

    if high_variance:
        # Penalize confidence for inconsistent signals
        variance_penalty = min(0.3, (score_sd - 3.0) * 0.1)
        final_confidence = max(0.1, base_confidence - variance_penalty)
    else:
        final_confidence = base_confidence

    return {
        "score": round(aggregated_score, 2),
        "confidence": round(final_confidence, 2),
        "variance": round(variance, 2),
        "item_count": len(scores),
        "method": method,
        "high_variance_flag": high_variance
    }
```

**Handling High Variance:**

When scores for the same sin vary substantially across questions (SD > 3.0), this could indicate:

| Cause | Implication | System Response |
|---|---|---|
| Genuine complexity | Person behaves differently in different contexts | Retain score; flag for downstream weighting |
| Measurement error | Questions elicited unreliable signals | Reduce confidence; flag for review |
| Response style | User answered inconsistently | Investigate at profile level |

The system cannot distinguish these causes from the data alone. High-variance flags propagate to similarity calculations, where flagged traits receive reduced weight.

---

## 5.4 Outlier Detection for Ultra-Short Scales

Traditional outlier detection methods (IQR, z-score, Mahalanobis distance) require 15-20+ data points to establish stable statistical estimates. With only 6 scores per trait, these methods are statistically invalid.

Research on ultra-short scales recommends **simple flagging rules** that can be implemented immediately, with more sophisticated IRT-based methods added as data accumulates.

**Tier 1: Simple Flags (Implementable at Launch)**

| Flag | Detection Method | Rationale |
|---|---|---|
| Zero variance | All 6 scores identical for a trait | Psychologically implausible; suggests disengagement |
| Extreme score | Any single score at ±5 with confidence > 0.9 | Verify evidence supports extreme rating |
| Score reversal | Adjacent questions: score flips from +4 to -4 | Possible parsing error or contradictory responses |

```
def detect_trait_outliers(scores: list[dict], sin_name: str) -> list[dict]:
    """
    Detect outlier patterns within a single trait's scores.

    Returns list of flags (empty if no issues detected).
    """
    flags = []
    values = [s["score"] for s in scores]
    confidences = [s["confidence"] for s in scores]

    # Zero variance check
    if len(set(values)) == 1 and len(values) >= 4:
        flags.append({
            "type": "zero_variance",
            "trait": sin_name,
            "severity": "warning",
            "detail": f"All {len(values)} scores identical: {values[0]}"
        })

    # Extreme score check
    for i, (v, c) in enumerate(zip(values, confidences)):
        if abs(v) == 5 and c > 0.9:
            flags.append({
                "type": "extreme_score",
                "trait": sin_name,
                "severity": "review",
                "detail": f"Q{i+1}: score={v}, confidence={c}"
            })

    # Score reversal check (adjacent questions)
    for i in range(len(values) - 1):
        if abs(values[i] - values[i+1]) >= 8:  # e.g., +4 to -4
```

```
            flags.append({
                "type": "score_reversal",
                "trait": sin_name,
                "severity": "warning",
                "detail": f"Q{i+1}→Q{i+2}: {values[i]}→{values[i+1]}"
            })

    return flags
```

**Tier 2: Statistical Thresholds (After N≥100 Profiles)**

Once baseline data exists, calculate population-level statistics and flag individual profiles that deviate:

```
def detect_statistical_outliers(
    trait_score: float,
    population_mean: float,
    population_sd: float
) -> dict | None:
    """
    Flag scores that deviate significantly from population norms.
    Requires pre-computed population statistics (N≥100).
    """
    if population_sd == 0:
        return None

    z_score = (trait_score - population_mean) / population_sd

    if abs(z_score) > 2.5:
        return {
            "type": "population_outlier",
            "z_score": round(z_score, 2),
            "severity": "review"
        }

    return None
```

**Tier 3: IRT Person-Fit (After N≥300 Profiles)**

With sufficient data, calibrate item response theory models and implement person-fit statistics (lz, U3). This is beyond launch scope but should be planned for.

---

## 5.5 Response Style Detection

Response styles are systematic tendencies to prefer certain response categories regardless of content. With only 6 items, detection capability is limited but still valuable.

**Extreme Response Style (ERS):**

Definition: Tendency to select scale endpoints (±4 or ±5) regardless of item content.

```
def detect_ers(all_scores: list[list[dict]]) -> dict:
    """
    Detect Extreme Response Style across all trait scores.

    Args:
        all_scores: 7 lists (one per sin), each containing up to 6 score dicts

    Returns:
        {"flag": bool, "percentage": float, "detail": str}
    """
    total_responses = 0
    extreme_responses = 0

    for sin_scores in all_scores:
        for score_dict in sin_scores:
            total_responses += 1
            if abs(score_dict["score"]) >= 4:
                extreme_responses += 1

    if total_responses == 0:
        return {"flag": False, "percentage": 0.0, "detail": "No responses"}

    percentage = extreme_responses / total_responses

    # Research threshold: >40% extreme responses
    flag = percentage > 0.40

    return {
        "flag": flag,
        "percentage": round(percentage * 100, 1),
        "detail": f"{extreme_responses}/{total_responses} responses at ±4 or ±5"
    }
```

**Midpoint Response Style (MRS):**

Definition: Preference for neutral/middle responses (scores near 0), potentially indicating disengagement.

```
def detect_mrs(all_scores: list[list[dict]], response_time_seconds: float, median_time: float) -> dict:
    """
    Detect Midpoint Response Style.

    MRS is only flagged when combined with fast completion time,
    to distinguish disengagement from genuine neutrality.
    """
    total_responses = 0
    midpoint_responses = 0

    for sin_scores in all_scores:
        for score_dict in sin_scores:
            total_responses += 1
            if abs(score_dict["score"]) <= 1:  # -1, 0, or +1
                midpoint_responses += 1

    if total_responses == 0:
        return {"flag": False, "percentage": 0.0, "detail": "No responses"}

    percentage = midpoint_responses / total_responses
    fast_completion = response_time_seconds < (median_time * 0.5)

    # Research threshold: >50% midpoints AND fast completion
    flag = percentage > 0.50 and fast_completion

    return {
        "flag": flag,
        "percentage": round(percentage * 100, 1),
        "fast_completion": fast_completion,
        "detail": f"{midpoint_responses}/{total_responses} responses near midpoint"
    }
```

**Pattern Detection:**

Certain response patterns suggest careless or automated responses:

```
def detect_response_patterns(all_scores: list[list[dict]]) -> dict:
    """
    Detect suspicious response patterns across questions.
    """
    flags = []

    # Collect scores by question (across all sins)
    # all_scores is [sin][question], we need [question][sin]
    num_questions = len(all_scores[0]) if all_scores else 0

    for q_idx in range(num_questions):
        question_scores = [sin_scores[q_idx]["score"] for sin_scores in all_scores if q_idx <
len(sin_scores)]

        # Check if all 7 sins got identical score for this question
        if len(set(question_scores)) == 1 and len(question_scores) == 7:
            flags.append({
                "type": "uniform_question",
                "question": q_idx + 1,
                "score": question_scores[0],
                "detail": f"Q{q_idx + 1}: All 7 sins scored {question_scores[0]}"
            })

    # Check for alternating pattern within each sin
    for sin_idx, sin_scores in enumerate(all_scores):
        values = [s["score"] for s in sin_scores]
        if len(values) >= 4:
            # Check if alternating (e.g., +2, -2, +2, -2)
            diffs = [values[i+1] - values[i] for i in range(len(values)-1)]
            if len(diffs) >= 3:
                alternating = all(
                    diffs[i] * diffs[i+1] < 0  # Signs alternate
                    for i in range(len(diffs)-1)
                )
                if alternating and all(abs(d) >= 3 for d in diffs):
                    flags.append({
                        "type": "alternating_pattern",
                        "sin_index": sin_idx,
                        "detail": f"Alternating extreme scores detected"
                    })

    return {
        "flag": len(flags) > 0,
        "patterns": flags
    }
```

## 5.6 Profile Quality Score

Research indicates no standardised "profile quality" metric exists for ultra-short scales. Harmonia implements a composite score combining four components, each normalised to 0-100.

**Component Metrics:**

| Component | What It Measures | Calculation |
|---|---|---|
| Internal Consistency | Do scores cohere within traits? | Mean confidence across all scores |
| Response Variance | Is there appropriate differentiation? | Penalise if too low or too high |
| Response Style | Are systematic biases present? | Deduct for ERS, MRS, pattern flags |
| Engagement | Did user invest effort? | Based on word count and response time |

**Quality Score Formula:**

```
def calculate_profile_quality(
    all_trait_results: dict,
    response_style_flags: dict,
    total_word_count: int,
    response_time_seconds: float,
    median_response_time: float = 300.0  # 5 minutes default until calibrated
) -> dict:
    """
    Calculate composite profile quality score (0-100).

    Returns:
        {
            "score": float,  # 0-100
            "tier": str,     # "high", "moderate", "low"
            "components": dict,
            "recommendation": str
        }
    """

    # Component 1: Internal Consistency (based on mean confidence)
    all_confidences = []
    for sin_name, trait_data in all_trait_results.items():
```

```
    all_confidences.append(trait_data["confidence"])

mean_confidence = statistics.mean(all_confidences) if all_confidences else 0
# Normalise: confidence 0.7 = 100, confidence 0.35 = 50, confidence 0 = 0
consistency_score = min(100, (mean_confidence / 0.7) * 100)

# Component 2: Response Variance
all_variances = [t["variance"] for t in all_trait_results.values()]
mean_variance = statistics.mean(all_variances) if all_variances else 0

# Optimal variance range: 1.0-6.0 on 11-point scale
if 1.0 <= mean_variance <= 6.0:
    variance_score = 100
elif mean_variance < 1.0:
    # Too little variance (possible disengagement)
    variance_score = max(30, mean_variance / 1.0 * 100)
else:
    # Too much variance (possible careless responding)
    variance_score = max(30, 100 - ((mean_variance - 6.0) * 10))

# Component 3: Response Style
style_flags_count = sum([
    1 if response_style_flags.get("ers", {}).get("flag", False) else 0,
    1 if response_style_flags.get("mrs", {}).get("flag", False) else 0,
    1 if response_style_flags.get("patterns", {}).get("flag", False) else 0
])
style_score = 100 - (style_flags_count * 25)

# Component 4: Engagement
# Word count component (expect 150-600 total for 6 questions)
word_score = min(100, (total_word_count / 300) * 100)

# Time component
time_ratio = response_time_seconds / median_response_time
if time_ratio < 0.3:
    time_score = 30  # Too fast
elif time_ratio > 3.0:
    time_score = 70  # Very slow (might indicate distraction, but not penalise heavily)
else:
    time_score = 100

engagement_score = (word_score + time_score) / 2

# Composite: equal-weighted average
```

```
composite = (consistency_score + variance_score + style_score + engagement_score) / 4

# Determine tier
if composite >= 80:
    tier = "high"
    recommendation = "Profile suitable for matching"
elif composite >= 60:
    tier = "moderate"
    recommendation = "Profile usable with caution; consider weighting down in matching"
else:
    tier = "low"
    recommendation = "Profile quality insufficient; recommend re-assessment or exclusion"

return {
    "score": round(composite, 1),
    "tier": tier,
    "components": {
        "consistency": round(consistency_score, 1),
        "variance": round(variance_score, 1),
        "style": round(style_score, 1),
        "engagement": round(engagement_score, 1)
    },
    "recommendation": recommendation
}
```

**Quality Tiers and Actions:**

| Score | Tier | Action |
|---|---|---|
| ≥80 | High | Full weight in matching; no restrictions |
| 60-79 | Moderate | Include in matching; apply 0.8× weight multiplier |
| <60 | Low | Exclude from matching OR flag for manual review |

---

## 5.7 Sin-Specific Weights

Weights apply to **similarity calculations**, not profile aggregation. A user's Envy score is computed the same way as their Wrath score; the difference is how much each contributes to match percentages.

**Weight Rationale:**

| Sin | Weight | Research Basis |
|---|---|---|
| Wrath | 1.5× | Conflict style is the strongest predictor of relationship satisfaction. Gottman's research identifies criticism, contempt, and defensiveness as primary predictors of relationship failure. |
| Sloth | 1.3× | Motivation and energy alignment affects shared activity participation. Mismatched activity levels create friction in daily life. |
| Pride | 1.2× | Ego dynamics affect communication quality. Status-seeking vs. humility differences can create resentment. |
| Lust | 1.0× | Baseline weight. Spontaneity preferences matter but are more context-dependent. |
| Greed | 0.9× | Resource attitudes matter for long-term compatibility but are less salient in early dating. |
| Gluttony | 0.8× | Lifestyle compatibility signal; indulgence preferences are visible but not relationship-critical. |
| Envy | 0.7× | Comparison orientation affects individual satisfaction but has less direct impact on dyadic interaction. |

**Weight Application:**

Weights are stored in configuration and applied during similarity calculation (Section 6), not during profile construction:

```
SIN_WEIGHTS = {
    "wrath": 1.5,
    "sloth": 1.3,
    "pride": 1.2,
    "lust": 1.0,
    "greed": 0.9,
    "gluttony": 0.8,
    "envy": 0.7
}

def get_weighted_traits(profile: dict) -> dict:
    """
    Return profile with weights attached (for similarity service).
    Does NOT modify scores; just attaches weight metadata.
    """
    weighted = {}
    for sin, data in profile["sins"].items():
        weighted[sin] = {
            **data,
            "weight": SIN_WEIGHTS[sin]
        }
    return weighted
```

---

## 5.8 Final Profile Structure

After all aggregation and quality checks, the profile is stored with complete metadata.

**Schema:**

```
{
    "user_id": "abc123",
    "version": 1,
    "created_at": "2025-01-29T14:30:00Z",
    "updated_at": "2025-01-29T14:30:00Z",

    "sins": {
        "greed": {
            "score": -1.2,
            "confidence": 0.72,
            "variance": 1.8,
            "item_count": 6,
            "method": "cwmv",
            "high_variance_flag": False
        },
        "pride": {
            "score": 0.5,
            "confidence": 0.58,
            "variance": 4.2,
            "item_count": 6,
            "method": "mean",
            "high_variance_flag": True
        },
        "lust": {
            "score": 2.8,
            "confidence": 0.81,
            "variance": 1.2,
            "item_count": 6,
            "method": "cwmv",
            "high_variance_flag": False
        },
    "wrath": {
      "score": -2.1,
      "confidence": 0.85,
      "variance": 0.9,
      "item_count": 6,
      "method": "cwmv",
      "high_variance_flag": False
    },
    "gluttony": {
      "score": 1.4,
      "confidence": 0.65,
      "variance": 2.1,
      "item_count": 6,
      "method": "mean",
      "high_variance_flag": False
    },
    "envy": {
      "score": -0.8,
      "confidence": 0.70,
      "variance": 1.5,
      "item_count": 6,
      "method": "cwmv",
      "high_variance_flag": False
    },
    "sloth": {
      "score": -1.5,
      "confidence": 0.77,
      "variance": 1.1,
      "item_count": 6,
      "method": "cwmv",
      "high_variance_flag": False
    }
  },

  "quality": {
    "score": 78.5,
    "tier": "moderate",
    "components": {
      "consistency": 82.0,
      "variance": 85.0,
      "style": 75.0,
      "engagement": 72.0
    },
    "recommendation": "Profile usable with caution"
    },

    "response_styles": {
        "ers": {"flag": False, "percentage": 18.5},
        "mrs": {"flag": False, "percentage": 22.0, "fast_completion": False},
        "patterns": {"flag": True, "patterns": [{"type": "uniform_question", "question": 3}]}
    },

    "flags": [
        "pride_high_variance",
        "q3_uniform_scores"
    ],

    "metadata": {
        "questions_answered": 6,
        "questions_valid": 6,
        "total_word_count": 487,
        "response_time_seconds": 342,
        "gemini_model": "gemini-3-pro-preview",
        "parsing_version": "1.0.0"
    }
}
```

**Profile Versioning:**

When a user retakes the questionnaire, the system creates a new profile version rather than modifying the existing one:

```
def handle_profile_update(user_id: str, new_profile: dict, db) -> dict:
    """
    Handle re-assessment: create new version, archive old.
    """
    existing = db.get_current_profile(user_id)

    if existing:
        # Archive old profile
        db.archive_profile(existing["profile_id"])

        # Increment version
        new_profile["version"] = existing["version"] + 1
    else:
        new_profile["version"] = 1

    new_profile["user_id"] = user_id
    new_profile["created_at"] = datetime.utcnow().isoformat()
    new_profile["updated_at"] = new_profile["created_at"]

    return db.save_profile(new_profile)
```

This preserves history for potential analysis while ensuring matching always uses the most recent assessment.

---

## 5.9 Complete ProfileService Implementation

The following service class integrates all aggregation logic:

```
"""
Profile aggregation service for Harmonia personality system.

Transforms GeminiService outputs into quality-controlled user profiles.
"""

import statistics
from datetime import datetime
from typing import Optional
import logging

logger = logging.getLogger(__name__)


class ProfileService:
    """
    Aggregates parsed question responses into a single personality profile.

    Pipeline:
    1. Validate individual Q&A pairs
    2. Aggregate scores per trait (CWMV or mean)
    3. Detect outliers and response styles
    4. Calculate quality score
    5. Compile final profile with metadata
    """

    SIN_NAMES = ["greed", "pride", "lust", "wrath", "gluttony", "envy", "sloth"]

    SIN_WEIGHTS = {
        "wrath": 1.5,
        "sloth": 1.3,
        "pride": 1.2,
        "lust": 1.0,
        "greed": 0.9,
        "gluttony": 0.8,
        "envy": 0.7
    }

    # Quality thresholds
    MIN_QUESTIONS = 4
    MIN_WORD_COUNT = 25
    LOW_WORD_COUNT = 50
    HIGH_VARIANCE_THRESHOLD = 3.0  # SD on 11-point scale
    ERS_THRESHOLD = 0.40
    MRS_THRESHOLD = 0.50

    def __init__(self, median_response_time: float = 300.0):
        """
        Args:
            median_response_time: Baseline for engagement scoring (seconds).
                                  Should be updated from population data.
        """
        self.median_response_time = median_response_time

    def build_profile(
        self,
        user_id: str,
        parsed_responses: list[dict],
        response_metadata: dict
    ) -> dict:
        """
        Build complete user profile from parsed question responses.

        Args:
            user_id: Unique user identifier
            parsed_responses: List of GeminiService outputs, one per question
                Each contains: {"sins": {sin_name: {"score", "confidence", "evidence"}}}
            response_metadata: {"word_counts": [...], "response_time_seconds": float}

        Returns:
            Complete profile dict ready for storage
        """
        logger.info(f"Building profile for user {user_id} from {len(parsed_responses)} responses")
```

```
        # Step 1: Validate questions
        valid_responses = []
        word_counts = response_metadata.get("word_counts", [0] * len(parsed_responses))

        for i, (parsed, word_count) in enumerate(zip(parsed_responses, word_counts)):
            validation = self._validate_question(parsed, word_count)
            if validation["valid"]:
                # Apply confidence penalty for short responses
                if validation["confidence_penalty"] > 0:
                    for sin in self.SIN_NAMES:
                        parsed["sins"][sin]["confidence"] = max(
                            0.1,
                            parsed["sins"][sin]["confidence"] - validation["confidence_penalty"]
                        )
                valid_responses.append(parsed)
            else:
                logger.warning(f"Q{i+1} excluded: {validation['exclusion_reason']}")

        # Check minimum questions
        if len(valid_responses) < self.MIN_QUESTIONS:
            logger.error(f"Insufficient valid responses:
{len(valid_responses)}/{self.MIN_QUESTIONS}")
            return self._create_rejected_profile(user_id, len(valid_responses))

        # Step 2: Organise scores by trait
        scores_by_trait = {sin: [] for sin in self.SIN_NAMES}

        for parsed in valid_responses:
            for sin in self.SIN_NAMES:
                scores_by_trait[sin].append(parsed["sins"][sin])

        # Step 3: Aggregate each trait
        aggregated_sins = {}
        all_flags = []

        for sin in self.SIN_NAMES:
            trait_result = self._aggregate_trait(scores_by_trait[sin])
            aggregated_sins[sin] = trait_result

            # Collect flags
            if trait_result["high_variance_flag"]:
                all_flags.append(f"{sin}_high_variance")

            # Check for outliers
```

```
    outlier_flags = self._detect_trait_outliers(scores_by_trait[sin], sin)
    for flag in outlier_flags:
        all_flags.append(f"{sin}_{flag['type']}")

# Step 4: Detect response styles
all_scores_list = [scores_by_trait[sin] for sin in self.SIN_NAMES]

ers_result = self._detect_ers(all_scores_list)
mrs_result = self._detect_mrs(
    all_scores_list,
    response_metadata.get("response_time_seconds", 300),
    self.median_response_time
)
pattern_result = self._detect_patterns(all_scores_list)

response_styles = {
    "ers": ers_result,
    "mrs": mrs_result,
    "patterns": pattern_result
}

if ers_result["flag"]:
    all_flags.append("ers_detected")
if mrs_result["flag"]:
    all_flags.append("mrs_detected")
if pattern_result["flag"]:
    all_flags.append("pattern_detected")

# Step 5: Calculate quality score
quality = self._calculate_quality(
    aggregated_sins,
    response_styles,
    sum(word_counts),
    response_metadata.get("response_time_seconds", 300)
)

# Step 6: Compile final profile
profile = {
    "user_id": user_id,
    "version": 1,
    "created_at": datetime.utcnow().isoformat(),
    "updated_at": datetime.utcnow().isoformat(),
    "sins": aggregated_sins,
    "quality": quality,
            "response_styles": response_styles,
            "flags": all_flags,
            "metadata": {
                "questions_answered": len(parsed_responses),
                "questions_valid": len(valid_responses),
                "total_word_count": sum(word_counts),
                "response_time_seconds": response_metadata.get("response_time_seconds"),
                "gemini_model": response_metadata.get("gemini_model", "unknown"),
                "parsing_version": "1.0.0"
            }
        }

        logger.info(
            f"Profile built for {user_id}: "
            f"quality={quality['score']:.1f} ({quality['tier']}), "
            f"flags={len(all_flags)}"
        )

        return profile

    def _validate_question(self, parsed: dict, word_count: int) -> dict:
        """Validate a single Q&A pair."""

        if word_count < self.MIN_WORD_COUNT:
            return {"valid": False, "confidence_penalty": 0, "exclusion_reason": "insufficient_words"}

        # Check for no signal
        all_scores = [parsed["sins"][sin]["score"] for sin in self.SIN_NAMES]
        all_confs = [parsed["sins"][sin]["confidence"] for sin in self.SIN_NAMES]

        if all(s == 0 for s in all_scores) and statistics.mean(all_confs) < 0.3:
            return {"valid": False, "confidence_penalty": 0, "exclusion_reason": "no_signal"}

        # Valid, possibly with penalty
        penalty = 0.2 if word_count < self.LOW_WORD_COUNT else 0
        return {"valid": True, "confidence_penalty": penalty, "exclusion_reason": None}

    def _aggregate_trait(self, scores: list[dict]) -> dict:
        """Aggregate multiple scores for one trait."""

        if not scores:
            return {
                "score": 0.0, "confidence": 0.0, "variance": 0.0,
                "item_count": 0, "method": "none", "high_variance_flag": False
            }

        values = [s["score"] for s in scores]
        confidences = [s["confidence"] for s in scores]

        variance = statistics.variance(values) if len(values) > 1 else 0.0
        conf_sd = statistics.stdev(confidences) if len(confidences) > 1 else 0.0

        # Choose aggregation method
        if conf_sd >= 0.10:
            weighted_sum = sum(v * c for v, c in zip(values, confidences))
            conf_sum = sum(confidences)
            score = weighted_sum / conf_sum if conf_sum > 0 else 0.0
            method = "cwmv"
        else:
            score = statistics.mean(values)
            method = "mean"

        # Calculate confidence with variance penalty
        base_conf = statistics.mean(confidences)
        score_sd = statistics.stdev(values) if len(values) > 1 else 0.0
        high_var = score_sd > self.HIGH_VARIANCE_THRESHOLD

        if high_var:
            penalty = min(0.3, (score_sd - self.HIGH_VARIANCE_THRESHOLD) * 0.1)
            final_conf = max(0.1, base_conf - penalty)
        else:
            final_conf = base_conf

        return {
            "score": round(score, 2),
            "confidence": round(final_conf, 2),
            "variance": round(variance, 2),
            "item_count": len(scores),
            "method": method,
            "high_variance_flag": high_var
        }

    def _detect_trait_outliers(self, scores: list[dict], sin_name: str) -> list[dict]:
        """Detect outlier patterns within one trait."""
        flags = []
        values = [s["score"] for s in scores]
        confidences = [s["confidence"] for s in scores]
```

```
        # Zero variance
        if len(set(values)) == 1 and len(values) >= 4:
            flags.append({"type": "zero_variance", "trait": sin_name, "severity": "warning"})

        # Extreme scores
        for i, (v, c) in enumerate(zip(values, confidences)):
            if abs(v) == 5 and c > 0.9:
                flags.append({"type": "extreme_score", "trait": sin_name, "severity": "review"})

        # Score reversals
        for i in range(len(values) - 1):
            if abs(values[i] - values[i+1]) >= 8:
                flags.append({"type": "score_reversal", "trait": sin_name, "severity": "warning"})

        return flags

    def _detect_ers(self, all_scores: list[list[dict]]) -> dict:
        """Detect Extreme Response Style."""
        total = 0
        extreme = 0

        for sin_scores in all_scores:
            for s in sin_scores:
                total += 1
                if abs(s["score"]) >= 4:
                    extreme += 1

        pct = extreme / total if total > 0 else 0
        return {
            "flag": pct > self.ERS_THRESHOLD,
            "percentage": round(pct * 100, 1),
            "detail": f"{extreme}/{total} at ±4 or ±5"
        }

    def _detect_mrs(self, all_scores: list[list[dict]], time: float, median: float) -> dict:
        """Detect Midpoint Response Style."""
        total = 0
        midpoint = 0

        for sin_scores in all_scores:
            for s in sin_scores:
                total += 1
                if abs(s["score"]) <= 1:
                    midpoint += 1
        pct = midpoint / total if total > 0 else 0
        fast = time < (median * 0.5)

        return {
            "flag": pct > self.MRS_THRESHOLD and fast,
            "percentage": round(pct * 100, 1),
            "fast_completion": fast,
            "detail": f"{midpoint}/{total} near midpoint"
        }

    def _detect_patterns(self, all_scores: list[list[dict]]) -> dict:
        """Detect suspicious response patterns."""
        patterns = []
        num_questions = len(all_scores[0]) if all_scores else 0

        for q_idx in range(num_questions):
            q_scores = [
                sin_scores[q_idx]["score"]
                for sin_scores in all_scores
                if q_idx < len(sin_scores)
            ]
            if len(set(q_scores)) == 1 and len(q_scores) == 7:
                patterns.append({"type": "uniform_question", "question": q_idx + 1})

        return {"flag": len(patterns) > 0, "patterns": patterns}

    def _calculate_quality(
        self,
        sins: dict,
        styles: dict,
        word_count: int,
        time: float
    ) -> dict:
        """Calculate composite quality score."""

        # Consistency
        confs = [s["confidence"] for s in sins.values()]
        mean_conf = statistics.mean(confs) if confs else 0
        consistency = min(100, (mean_conf / 0.7) * 100)

        # Variance
        vars_ = [s["variance"] for s in sins.values()]
        mean_var = statistics.mean(vars_) if vars_ else 0
```

```
if 1.0 <= mean_var <= 6.0:
    variance_score = 100
elif mean_var < 1.0:
    variance_score = max(30, mean_var / 1.0 * 100)
else:
    variance_score = max(30, 100 - ((mean_var - 6.0) * 10))

# Style
flags_count = sum([
    styles.get("ers", {}).get("flag", False),
    styles.get("mrs", {}).get("flag", False),
    styles.get("patterns", {}).get("flag", False)
])
style_score = 100 - (flags_count * 25)

# Engagement
word_score = min(100, (word_count / 300) * 100)
time_ratio = time / self.median_response_time
if time_ratio < 0.3:
    time_score = 30
elif time_ratio > 3.0:
    time_score = 70
else:
    time_score = 100
engagement = (word_score + time_score) / 2

# Composite
composite = (consistency + variance_score + style_score + engagement) / 4

if composite >= 80:
    tier, rec = "high", "Profile suitable for matching"
elif composite >= 60:
    tier, rec = "moderate", "Profile usable with caution"
else:
    tier, rec = "low", "Profile quality insufficient; recommend re-assessment"

return {
    "score": round(composite, 1),
    "tier": tier,
    "components": {
        "consistency": round(consistency, 1),
        "variance": round(variance_score, 1),
        "style": round(style_score, 1),
        "engagement": round(engagement, 1)
```

```
            },
            "recommendation": rec
        }

    def _create_rejected_profile(self, user_id: str, valid_count: int) -> dict:
        """Create a rejected profile when minimum requirements not met."""
        return {
            "user_id": user_id,
            "version": 1,
            "created_at": datetime.utcnow().isoformat(),
            "updated_at": datetime.utcnow().isoformat(),
            "sins": None,
            "quality": {
                "score": 0,
                "tier": "rejected",
                "components": None,
                "recommendation": f"Only {valid_count}/{self.MIN_QUESTIONS} valid responses"
            },
            "response_styles": None,
            "flags": ["insufficient_responses"],
            "metadata": {"questions_valid": valid_count}
        }
```

---

## 5.10 Summary

**Profile Aggregation Pipeline:**

| Stage | Input | Output | Key Decisions |
|---|---|---|---|
| Question Validation | 6 Q&A pairs + word counts | 4-6 valid pairs | Min 25 words; exclude no-signal responses |
| Trait Aggregation | Up to 6 scores per sin | 7 aggregated sin scores | CWMV if confidence varies; simple mean otherwise |
| Outlier Detection | Score distributions | Warning/review flags | Zero variance, extreme scores, reversals |
| Response Style | All 42 scores + timing | ERS/MRS/pattern flags | >40% extreme = ERS; >50% midpoint + fast = MRS |
| Quality Score | All components | 0-100 composite | ≥80 high, 60-79 moderate, <60 low |

| | | | |
|---|---|---|---|
| Profile Assembly | All above | Complete profile JSON | Versioned, timestamped, ready for storage |

**What This Achieves:**

1. **Quality transparency** — Every profile has a quality score and tier
2. **Graceful degradation** — Missing questions reduce confidence, don't crash the system
3. **Response style detection** — Systematic biases are flagged, not hidden
4. **Audit trail** — Flags and metadata explain why a profile scored as it did
5. **Future-proof structure** — Versioning and metadata support iteration

**What It Doesn't Achieve:**

1. **Ground-truth validation** — Quality score is heuristic until validated against matching outcomes
2. **IRT-based person-fit** — Requires N≥300 for calibration (Tier 3 roadmap)
3. **Cultural adjustment** — Thresholds assume Western response patterns
4. **Acquiescence detection** — Would require balanced items (not in current questionnaire)

**Implementation Phases:**

| Phase | Data Requirement | Capability |
|---|---|---|
| Launch | N/A | Tier 1 flags, heuristic quality score |
| Early Data (N≥100) | Population statistics | Adjusted thresholds, z-score outliers |
| Established (N≥300) | IRT calibration | Person-fit statistics, empirical weights |

## Section 6: Similarity Calculation

### 6.1 The Sequential Matching Architecture

Harmonia does not combine visual, personality, and genetic signals into a single weighted score. Instead, it uses a **cascaded filtering pipeline** where each stage gates access to the next. This architecture, validated by production systems at Kuaishou, Meta, and YouTube, outperforms weighted averaging by 30%+ on match conversion metrics.

**The Three-Stage Pipeline:**

```
┌─────────────────────────────────────────────────────────────┐
│ STAGE 1: VISUAL GATE (60% of matching decision)             │
│                                                             │
│ User A sees User B's photos → Swipes                        │
│ User B sees User A's photos → Swipes                        │
│                                                             │
│ Gate: MUTUAL SWIPE REQUIRED                                 │
│ If both swipe right → Proceed to Stage 2                    │
│ If either swipes left → No match; pipeline terminates       │
└─────────────────────────────────────────────────────────────┘
              ↓
     [Only if mutual interest]
              ↓
┌─────────────────────────────────────────────────────────────┐
│ STAGE 2: PERSONALITY REVEAL (30% of matching decision)      │
│                                                             │
│ Calculate perceived similarity between profiles             │
│ Display shared traits ("You're both...")                    │
│                                                             │
│ Gate: SIMILARITY THRESHOLD (configurable, default ≥0.40)    │
│ If threshold met → Proceed to Stage 3                       │
│ If below threshold → Match proceeds but flagged as "low fit"│
└─────────────────────────────────────────────────────────────┘
              ↓
     [Personality revealed to users]
              ↓
```

```
┌──────────────────────────────────────────────────────────────
───────┐
│ STAGE 3: GENETICS INFO (10% of matching decision)            │
│                                                  │
│ Display HLA compatibility score                        │
│ Show "chemistry" indicator                           │
│                                                  │
│ Gate: NONE — Informational only                          │
│ HLA score displayed but does not block matching               │
└──────────────────────────────────────────────────────────────
───────┘
                    ↓
          [Full match card displayed]
                    ↓
             MATCH COMPLETE
          Users can now message
```

**Why Sequential Beats Weighted Averaging:**

| Dimension | Weighted Formula | Cascaded Pipeline |
|---|---|---|
| Reciprocity | Cannot enforce (scores are unilateral) | Built-in via mutual swipe gate |
| User Psychology | Unnatural — humans don't think in weighted sums | Matches how people evaluate: photos first, then personality |
| Threshold Control | Global only | Per-stage, per-segment tuning |
| Computational Cost | Must calculate all signals for all pairs | Only calculate personality for mutual matches |
| Match Conversion | Baseline | +28-30% (FAIR-MATCH 2025, GRank 2025) |

**The 60/30/10 Distribution:**

These percentages represent **decision importance**, not formula weights:

- **60% Visual**: The mutual swipe gate eliminates most incompatible pairs. Users who don't find each other attractive never reach personality comparison.
- **30% Personality**: For pairs who pass the visual gate, perceived similarity determines match quality and conversation likelihood.

- **10% Genetics**: HLA compatibility provides a final "chemistry" signal but doesn't block matches. Science is emerging, not definitive.

---

## 6.2 The Perceived Similarity Principle

Harmonia's personality matching is built on a counterintuitive research finding: **perceived similarity predicts initial attraction; actual similarity does not.**

**Research Foundation:**

Tidwell, Eastwick & Finkel (2013) conducted speed-dating studies measuring both actual trait similarity (comparing Big Five scores) and perceived similarity (how similar participants believed they were). Results:

| Similarity Type | Correlation with Romantic Liking |
|---|---|
| Actual similarity | r = 0.01 (not significant) |
| Perceived similarity | r = 0.39 (highly significant) |

Montoya, Horton & Kirchner (2008) meta-analysis confirmed: perceived similarity effects are robust, while actual similarity effects disappear in real interactions.

**Implication for Harmonia:**

We don't need to find people who are *objectively* similar. We need to find people who will *perceive each other* as similar when their shared traits are highlighted. This is the "astrology effect" — showing people their commonalities creates a sense of connection, regardless of actual profile distance.

**Design Consequence: Positive Overlap Only**

The similarity algorithm:

- ✅ Counts traits where both users share the same direction (both virtuous OR both vice-leaning)
- ✅ Rewards stronger alignment within shared traits
- ❌ Does NOT penalise differences
- ❌ Does NOT count opposite-direction traits against the score

Two users who share 3 of 7 traits get credit for those 3. The 4 traits where they differ are simply not mentioned — to the users, it feels like they have "so much in common."

---

## 6.3 The Core Algorithm: Positive Overlap

### Step 1: Define "Shared Direction"

Two users share a trait when both scores fall on the same side of neutral:

| User A Score | User B Score | Shared? | Interpretation |
|---|---|---|---|
| +2.5 | +3.1 | ✅ Yes (both vice) | "You're both spontaneous" |
| -2.0 | -1.8 | ✅ Yes (both virtue) | "You're both conflict-avoidant" |
| +2.5 | -1.8 | ❌ No (opposite) | Not mentioned to users |
| +0.3 | +0.2 | ❌ No (both neutral) | Signal too weak to highlight |
| -0.4 | +1.2 | ❌ No (opposite sides) | Not mentioned to users |

**The Neutral Zone:**

Scores between -0.5 and +0.5 are treated as "no clear signal." They don't count as overlap even if both users fall in this range — there's nothing meaningful to highlight.

python
```
NEUTRAL_THRESHOLD = 0.5

def is_shared_direction(score_a: float, score_b: float) -> tuple[bool, str]:
    """
    Determine if two scores represent a shared trait direction.

    Returns:
        (is_shared: bool, direction: str or None)
    """
    # Check if either score is in neutral zone
    if abs(score_a) <= NEUTRAL_THRESHOLD or abs(score_b) <= NEUTRAL_THRESHOLD:
        return False, None

    # Check if same direction
    both_vice = score_a > NEUTRAL_THRESHOLD and score_b > NEUTRAL_THRESHOLD
    both_virtue = score_a < -NEUTRAL_THRESHOLD and score_b < -NEUTRAL_THRESHOLD

    if both_vice:
        return True, "vice"
    elif both_virtue:
        return True, "virtue"
    else:
        return False, None
```

**Step 2: Calculate Trait Similarity Strength**

When two users share a trait direction, we measure how closely aligned their scores are:
```
trait_similarity = 1 - (|score_a - score_b| / 10)
```

This yields a value from 0.0 to 1.0:
- Identical scores (+3.0 and +3.0): similarity = 1.0
- Close scores (+2.0 and +3.5): similarity = 0.85
- Distant but same direction (+1.0 and +4.5): similarity = 0.65

**Step 3: Apply Confidence Weighting**

Low-confidence scores contribute less to similarity:
```
weighted_similarity = trait_similarity × avg_confidence
```

Where `avg_confidence = (confidence_a + confidence_b) / 2`

**Step 4: Apply Sin-Specific Weights**

As established in Section 5.7, different sins matter more for relationship compatibility:

```python
SIN_WEIGHTS = {
    "wrath": 1.5,    # Conflict style most important
    "sloth": 1.3,    # Motivation alignment
    "pride": 1.2,    # Ego dynamics
    "lust": 1.0,     # Baseline
    "greed": 0.9,    # Resource attitudes
    "gluttony": 0.8, # Moderation
    "envy": 0.7      # Comparison (lowest)
}
```

**Step 5: Calculate Final Similarity Score**
```
similarity = Σ(trait_similarity × avg_confidence × sin_weight) / Σ(sin_weight)
```

The denominator (sum of all sin weights = 7.4) normalises the score to a 0-1 range.

---

## 6.4 Worked Example

**User A Profile:**

python

```
{
    "greed":    {"score": -1.8, "confidence": 0.75},  # Generous
    "pride":    {"score": +0.3, "confidence": 0.60},  # Neutral
    "lust":     {"score": +2.5, "confidence": 0.82},  # Spontaneous
    "wrath":    {"score": -2.1, "confidence": 0.88},  # Conflict-avoidant
    "gluttony": {"score": +1.2, "confidence": 0.65},  # Slightly indulgent
    "envy":     {"score": -0.8, "confidence": 0.70},  # Content
    "sloth":    {"score": -1.5, "confidence": 0.78}   # Proactive
}
```

**User B Profile:**

python

```
{
    "greed":    {"score": -2.2, "confidence": 0.80},  # Generous
    "pride":    {"score": -1.5, "confidence": 0.72},  # Humble
    "lust":     {"score": +3.1, "confidence": 0.85},  # Spontaneous
    "wrath":    {"score": -1.8, "confidence": 0.82},  # Conflict-avoidant
    "gluttony": {"score": -0.3, "confidence": 0.55},  # Neutral
    "envy":     {"score": -1.2, "confidence": 0.68},  # Content
    "sloth":    {"score": +1.8, "confidence": 0.72}   # Slightly passive
}
```

**Step-by-Step Calculation:**

| Sin | A Score | B Score | Shared? | Why | Trait Sim | Avg Conf | Weight | Contribution |
|-----|---------|---------|---------|-----|-----------|----------|--------|--------------|
| Greed | -1.8 | -2.2 | ✅ Both virtue | Both generous | 0.96 | 0.775 | 0.9 | 0.670 |
| Pride | +0.3 | -1.5 | ❌ | A is neutral | — | — | — | 0 |
| Lust | +2.5 | +3.1 | ✅ Both vice | Both spontaneous | 0.94 | 0.835 | 1.0 | 0.785 |
| Wrath | -2.1 | -1.8 | ✅ Both virtue | Both conflict-avoidant | 0.97 | 0.850 | 1.5 | 1.237 |
| Gluttony | +1.2 | -0.3 | ❌ | B is neutral | — | — | — | 0 |
| Envy | -0.8 | -1.2 | ✅ Both virtue | Both content | 0.96 | 0.690 | 0.7 | 0.464 |
| Sloth | -1.5 | +1.8 | ❌ | Opposite directions | — | — | — | 0 |

~~~
**Totals:**
```
Sum of contributions: 0.670 + 0.785 + 1.237 + 0.464 = 3.156
Sum of weights (max possible): 7.4

Similarity score: 3.156 / 7.4 = 0.426 (42.6%)
~~~

**Interpretation:**

User A and User B share 4 of 7 traits (Greed, Lust, Wrath, Envy). Their strongest alignment is on conflict avoidance (Wrath), which carries the highest weight. Despite differences on Pride, Gluttony, and Sloth, they would perceive each other as moderately similar.

The match card would display:

> **You're both...**
>
> - Direct about money and generous with friends
> - Spontaneous and up for adventures
> - Prefer harmony over confrontation
> - Content with what you have

The differences (Pride, Gluttony, Sloth) are simply not mentioned.

---

## 6.5 Quality-Adjusted Similarity

Profile quality (Section 5.6) affects how much weight a similarity calculation receives. When one or both profiles have quality issues, the similarity score is discounted.

**Quality Multipliers:**

| Profile A Quality | Profile B Quality | Similarity Multiplier |
|---|---|---|
| High (≥80) | High (≥80) | 1.0 (full weight) |
| High (≥80) | Moderate (60-79) | 0.9 |
| Moderate (60-79) | Moderate (60-79) | 0.8 |
| High (≥80) | Low (<60) | 0.7 |
| Moderate (60-79) | Low (<60) | 0.6 |
| Low (<60) | Low (<60) | 0.5 |

**Rationale:**

A similarity score between two high-quality profiles is more trustworthy than one between two low-quality profiles. The multiplier ensures that matches built on solid data receive full credit, while matches built on shaky data are treated with appropriate skepticism.

python
```
def get_quality_multiplier(quality_a: float, quality_b: float) -> float:
    """
    Calculate similarity multiplier based on profile quality scores.
    """
    def tier(q):
        if q >= 80:
            return "high"
        elif q >= 60:
            return "moderate"
        else:
            return "low"

    tier_a = tier(quality_a)
    tier_b = tier(quality_b)

    multipliers = {
        ("high", "high"): 1.0,
        ("high", "moderate"): 0.9,
        ("moderate", "high"): 0.9,
        ("moderate", "moderate"): 0.8,
        ("high", "low"): 0.7,
        ("low", "high"): 0.7,
        ("moderate", "low"): 0.6,
        ("low", "moderate"): 0.6,
        ("low", "low"): 0.5
    }

    return multipliers.get((tier_a, tier_b), 0.5)
```

**Adjusted Similarity:**
```

adjusted_similarity = raw_similarity × quality_multiplier
```

From the previous example:

- If both profiles are high quality (≥80): 0.426 × 1.0 = **0.426**

- If A is high, B is moderate: 0.426 × 0.9 = **0.383**
- If both are low quality: 0.426 × 0.5 = **0.213**

---

## 6.6 Stage 2 Threshold Configuration

The personality similarity score determines whether a match is highlighted or flagged. The threshold is configurable and should be optimised per user segment through A/B testing.

**Default Threshold: 0.40**

Research on cascaded systems recommends starting conservative and adjusting based on match outcomes:

| Threshold | Effect | Trade-off |
|---|---|---|
| 0.30 | Very permissive | More matches, lower average quality |
| 0.40 | **Default** | Balanced coverage and quality |
| 0.50 | Selective | Fewer matches, higher quality signal |
| 0.60 | Very selective | May frustrate users with few matches |

**Threshold Actions:**

python
```
def evaluate_stage2(similarity: float, threshold: float = 0.40) -> dict:
    """
    Evaluate whether a match passes the personality stage.

    Returns:
        {
            "passed": bool,
            "tier": str,
            "display_mode": str
        }
    """
    if similarity >= threshold + 0.20:
        return {
            "passed": True,
            "tier": "strong_fit",
            "display_mode": "highlight"  # Emphasise shared traits
        }
    elif similarity >= threshold:
        return {
            "passed": True,
            "tier": "good_fit",
            "display_mode": "standard"  # Show shared traits normally
        }
    elif similarity >= threshold - 0.15:
        return {
            "passed": True,
            "tier": "moderate_fit",
            "display_mode": "minimal"  # Show fewer details
        }
    else:
        return {
            "passed": True,  # Still allow match — soft gate
            "tier": "low_fit",
            "display_mode": "chemistry_focus"  # Emphasise HLA instead

        }
```

**Important:** Stage 2 is a **soft gate**. Even low-similarity pairs can match — they just see different messaging. The threshold affects *presentation*, not *permission*.

---

## 6.7 Generating Match Explanations

The "astrology effect" requires translating overlap data into natural language that feels personal and insightful.

**Sin-to-Description Mapping:**

python
```
TRAIT_DESCRIPTIONS = {
    "greed": {
        "virtue": {
            "short": "Generous",
            "shared": "You're both generous and don't sweat the small stuff about money"
        },
        "vice": {
            "short": "Practical about money",
            "shared": "You're both practical and think carefully about resources"
        }
    },
    "pride": {
```

```
    "virtue": {
        "short": "Down-to-earth",
        "shared": "You're both humble and don't need the spotlight"
    },
    "vice": {
        "short": "Confident",
        "shared": "You both know your worth and aren't afraid to show it"
    }
},
"lust": {
    "virtue": {
        "short": "Thoughtful",
        "shared": "You both think things through before diving in"
    },
    "vice": {
        "short": "Spontaneous",
        "shared": "You're both spontaneous and up for adventure"
    }
},
"wrath": {
    "virtue": {
        "short": "Easygoing",
        "shared": "You both prefer harmony and avoid unnecessary drama"
    },
    "vice": {
        "short": "Direct",
        "shared": "You're both direct and don't shy away from tough conversations"
    }
},
"gluttony": {
    "virtue": {
        "short": "Balanced",
        "shared": "You both believe in moderation and taking care of yourselves"
    },
    "vice": {
        "short": "Fun-loving",
        "shared": "You both know how to enjoy the good things in life"
    }
},
"envy": {
    "virtue": {
        "short": "Content",
        "shared": "You're both secure in yourselves and don't compare to others"
    },
        "vice": {
            "short": "Ambitious",
            "shared": "You're both driven and motivated by success"
        }
    },
    "sloth": {
        "virtue": {
            "short": "Proactive",
            "shared": "You both take initiative and get things done"
        },
        "vice": {
            "short": "Relaxed",
            "shared": "You both value downtime and don't stress about productivity"
        }
    }
}
```

**Explanation Generator:**

python
```python
def generate_match_explanation(
    overlap_breakdown: list[dict],
    similarity_score: float,
    display_mode: str
) -> dict:
    """
    Generate user-facing match explanation from overlap data.

    Args:
        overlap_breakdown: List of {"sin", "direction", "contribution"}
        similarity_score: 0-1 score
        display_mode: "highlight", "standard", "minimal", "chemistry_focus"

    Returns:
        {
            "headline": str,
            "shared_traits": list[str],
            "summary": str
        }
    """
    # Sort by contribution (highest first)
    sorted_overlap = sorted(
        overlap_breakdown,
        key=lambda x: x["contribution"],
        reverse=True
    )

    # Determine how many traits to show
    if display_mode == "highlight":
        max_traits = 4
    elif display_mode == "standard":
        max_traits = 3
    elif display_mode == "minimal":
        max_traits = 2
    else:  # chemistry_focus
        max_traits = 1

    # Build shared trait descriptions
    shared_traits = []
    for item in sorted_overlap[:max_traits]:
        sin = item["sin"]
        direction = item["direction"]
        description = TRAIT_DESCRIPTIONS[sin][direction]["shared"]
        shared_traits.append(description)

    # Generate headline
    if similarity_score >= 0.60:
        headline = "Strong personality match!"
    elif similarity_score >= 0.45:
        headline = "You have a lot in common"
    elif similarity_score >= 0.30:
        headline = "Some shared traits"
    else:
        headline = "Different perspectives, potential spark"

    # Generate summary
    trait_count = len(sorted_overlap)
    if trait_count >= 5:
        summary = f"You align on {trait_count} personality traits"
    elif trait_count >= 3:
        summary = f"You share {trait_count} key traits"
    elif trait_count >= 1:
        summary = "You have some things in common"
    else:
        summary = "Your chemistry might surprise you"

    return {
        "headline": headline,
        "shared_traits": shared_traits,
        "summary": summary,
        "overlap_count": trait_count,
        "total_traits": 7
    }
```

**Example Output:**

For the User A / User B example (similarity = 0.426, 4 shared traits):

python
```
{
    "headline": "You have a lot in common",
    "shared_traits": [
        "You both prefer harmony and avoid unnecessary drama",
        "You're both spontaneous and up for adventure",
        "You're both generous and don't sweat the small stuff about money",
        "You're both secure in yourselves and don't compare to others"
    ],
    "summary": "You share 4 key traits",
    "overlap_count": 4,
    "total_traits": 7
}
```

---

## 6.8 Integration with HLA (Stage 3)

After personality is revealed, HLA compatibility provides the final signal. As established by research, this is **informational only** — it enhances the match card but doesn't gate anything.

**HLA Display Logic:**

python
```
def generate_hla_display(hla_score: float | None) -> dict:
    """
    Generate HLA compatibility display for match card.

    Args:
        hla_score: 0-100 chemistry score, or None if user hasn't uploaded genetics

    Returns:
        Display configuration for match card
    """
    if hla_score is None:
        return {
            "show": False,
            "reason": "genetics_not_uploaded"
        }

    if hla_score >= 75:
        return {
            "show": True,
            "icon": "🔥",
            "label": "Strong chemistry signal",
            "detail": "Your genetics suggest natural attraction"
        }
    elif hla_score >= 50:
        return {
            "show": True,
            "icon": "✨",
            "label": "Good chemistry",
            "detail": "Solid genetic compatibility"
        }
    elif hla_score >= 25:
        return {
            "show": True,
            "icon": "💫",
            "label": "Some chemistry",
            "detail": "Moderate genetic compatibility"
        }
    else:
        return {
            "show": False,  # Don't display low scores — no negative messaging
            "reason": "score_below_display_threshold"

        }
```

**Design Principle:** Never show negative HLA results. If compatibility is low, simply don't mention it. The user sees personality overlap and assumes genetics are neutral or positive.

---

### 6.9 Complete Match Card Assembly

The full match card combines all stages:

python
```
def assemble_match_card(
    user_a_id: str,
    user_b_id: str,
    similarity_result: dict,
    hla_score: float | None,
    profile_b: dict
) -> dict:
    """
    Assemble complete match card for display after mutual swipe.

    Returns:
        Complete match card data structure
    """
    # Get personality explanation
    explanation = generate_match_explanation(
        similarity_result["breakdown"],
        similarity_result["adjusted_score"],
        similarity_result["display_mode"]
    )

    # Get HLA display
    hla_display = generate_hla_display(hla_score)

    # Assemble card
    match_card = {
        "match_id": f"{user_a_id}_{user_b_id}",
        "matched_at": datetime.utcnow().isoformat(),

        # Basic info
        "user": {
            "id": user_b_id,
            "name": profile_b["display_name"],
            "photos": profile_b["photos"][:3],  # Top 3 photos
            "age": profile_b["age"],
            "location": profile_b["location_display"]
        },

        # Personality section
        "personality": {
            "headline": explanation["headline"],
            "shared_traits": explanation["shared_traits"],
            "summary": explanation["summary"],
            "score_display": f"{int(similarity_result['adjusted_score'] * 100)}%",
            "overlap_count": explanation["overlap_count"]
        },
        # Chemistry section (if available)
        "chemistry": hla_display if hla_display["show"] else None,

        # Metadata
        "match_quality": {
            "personality_tier": similarity_result["tier"],
            "profile_quality": similarity_result["quality_tier"]
        }
    }

    return match_card
```

**Example Match Card JSON:**

json

```
{
    "match_id": "abc123_def456",
    "matched_at": "2026-01-29T15:30:00Z",
    "user": {
        "id": "def456",
        "name": "Sarah",
        "photos": ["url1", "url2", "url3"],
        "age": 28,
        "location": "London"
    },
    "personality": {
        "headline": "You have a lot in common",
        "shared_traits": [
            "You both prefer harmony and avoid unnecessary drama",
            "You're both spontaneous and up for adventure",
            "You're both generous and don't sweat the small stuff about money"
        ],
        "summary": "You share 4 key traits",
        "score_display": "43%",
        "overlap_count": 4
    },
    "chemistry": {
        "show": true,
        "icon": "🔥",
        "label": "Strong chemistry signal",
        "detail": "Your genetics suggest natural attraction"
    },
    "match_quality": {
        "personality_tier": "good_fit",
        "profile_quality": "high"
    }
}
```

---

## 6.10 Edge Cases and Boundary Conditions

**No Overlapping Traits:**

When two users have zero shared-direction traits:

python
```
if len(overlap_breakdown) == 0:
    return {
        "adjusted_score": 0.0,
        "tier": "no_overlap",
        "display_mode": "chemistry_focus",
        "explanation": {
            "headline": "Different perspectives, potential spark",
            "shared_traits": [],
            "summary": "You might balance each other out"
        }
    }
```

The match still proceeds (soft gate), but messaging pivots to chemistry/attraction rather than personality fit.

**One Profile Rejected/Low Quality:**

When one profile has quality tier "rejected":

python
```
if profile_a["quality"]["tier"] == "rejected" or profile_b["quality"]["tier"] == "rejected":
    return {
        "adjusted_score": 0.0,
        "tier": "insufficient_data",
        "display_mode": "minimal",
        "explanation": {
            "headline": "Match!",
            "shared_traits": [],
            "summary": "Get to know each other"
        }
    }
```

No personality data is shown — the match card focuses on photos and basic info.

**Missing HLA Data:**

When either user hasn't uploaded genetic data:

python
```
if hla_score is None:
    # Simply omit chemistry section
    match_card["chemistry"] = None
```

The match card displays personality without the chemistry section. No negative messaging about missing data.

**Perfect Overlap:**

Theoretical maximum (all 7 traits shared with identical scores and perfect confidence):

```
max_score = (1.0 × 1.0 × 1.5) + (1.0 × 1.0 × 1.3) + ... = 7.4 / 7.4 = 1.0
```

In practice, scores above 0.70 are rare and indicate unusually strong alignment.

---

## 6.11 Complete SimilarityService Implementation

python
```
"""
Similarity calculation service for Harmonia matching system.

Implements perceived similarity using positive overlap detection.
Designed for Stage 2 of the cascaded matching pipeline.
"""

import statistics
from datetime import datetime
from typing import Optional
import logging

logger = logging.getLogger(__name__)
```

```python
class SimilarityService:
    """
    Calculate perceived similarity between two personality profiles.

    Key Principle: Focus on POSITIVE OVERLAP only.
    - Shared traits (both virtue OR both vice) contribute to similarity
    - Differences are ignored, not penalised
    - This creates the "astrology effect" where users feel connected
    """

    SIN_NAMES = ["greed", "pride", "lust", "wrath", "gluttony", "envy", "sloth"]

    SIN_WEIGHTS = {
        "wrath": 1.5,    # Conflict style most important
        "sloth": 1.3,    # Motivation alignment
        "pride": 1.2,    # Ego dynamics
        "lust": 1.0,     # Baseline
        "greed": 0.9,    # Resource attitudes
        "gluttony": 0.8, # Moderation
        "envy": 0.7      # Comparison (lowest)
    }

    TOTAL_WEIGHT = sum(SIN_WEIGHTS.values())  # 7.4

    NEUTRAL_THRESHOLD = 0.5  # Scores between -0.5 and +0.5 are neutral

    DEFAULT_STAGE2_THRESHOLD = 0.40

    TRAIT_DESCRIPTIONS = {
        "greed": {
            "virtue": "generous and easygoing about money",
            "vice": "practical and thoughtful about resources"
        },
        "pride": {
            "virtue": "humble and down-to-earth",
            "vice": "confident and self-assured"
        },
        "lust": {
            "virtue": "thoughtful and deliberate",
            "vice": "spontaneous and adventurous"
        },
        "wrath": {
            "virtue": "easygoing and harmony-seeking",
            "vice": "direct and unafraid of confrontation"
        },
        "gluttony": {
            "virtue": "balanced and moderate",
            "vice": "fun-loving and indulgent"
        },
        "envy": {
            "virtue": "content and secure",
            "vice": "ambitious and driven"
        },
        "sloth": {
            "virtue": "proactive and energetic",
            "vice": "relaxed and laid-back"
        }
    }

    def __init__(self, stage2_threshold: float = None):
        self.stage2_threshold = stage2_threshold or self.DEFAULT_STAGE2_THRESHOLD

    def calculate_similarity(
        self,
        profile_a: dict,
        profile_b: dict
    ) -> dict:
        """
        Calculate perceived similarity between two profiles.

        Args:
            profile_a: First user's profile (from ProfileService)
            profile_b: Second user's profile

        Returns:
            {
                "raw_score": float,
                "adjusted_score": float,
                "quality_multiplier": float,
                "breakdown": list[dict],
                "overlap_count": int,
                "tier": str,
                "display_mode": str
            }
        """
        logger.info(f"Calculating similarity: {profile_a['user_id']} ↔ {profile_b['user_id']}")

        # Handle rejected profiles
        if profile_a.get("quality", {}).get("tier") == "rejected" or \
           profile_b.get("quality", {}).get("tier") == "rejected":
            return self._insufficient_data_result(profile_a, profile_b)

        # Calculate raw similarity
        raw_score, breakdown = self._calculate_raw_similarity(
            profile_a["sins"],
            profile_b["sins"]
        )

        # Apply quality adjustment
        quality_a = profile_a.get("quality", {}).get("score", 70)
        quality_b = profile_b.get("quality", {}).get("score", 70)
        quality_multiplier = self._get_quality_multiplier(quality_a, quality_b)

        adjusted_score = raw_score * quality_multiplier

        # Determine tier and display mode
        tier, display_mode = self._evaluate_threshold(adjusted_score)

        # Determine quality tier for metadata
        quality_tier = self._combined_quality_tier(quality_a, quality_b)

        result = {
            "raw_score": round(raw_score, 4),
            "adjusted_score": round(adjusted_score, 4),
            "quality_multiplier": quality_multiplier,
            "breakdown": breakdown,
            "overlap_count": len(breakdown),
            "tier": tier,
            "display_mode": display_mode,
            "quality_tier": quality_tier
        }

        logger.info(
            f"Similarity result: raw={raw_score:.3f}, "
            f"adjusted={adjusted_score:.3f}, "
            f"overlap={len(breakdown)}/7, tier={tier}"
        )

        return result

    def _calculate_raw_similarity(
        self,
        sins_a: dict,
        sins_b: dict
    ) -> tuple[float, list[dict]]:
        """Calculate raw similarity score and breakdown."""

        total_contribution = 0.0
        breakdown = []

        for sin in self.SIN_NAMES:
            data_a = sins_a.get(sin, {"score": 0, "confidence": 0.5})
            data_b = sins_b.get(sin, {"score": 0, "confidence": 0.5})

            score_a = data_a.get("score", 0)
            score_b = data_b.get("score", 0)
            conf_a = data_a.get("confidence", 0.5)
            conf_b = data_b.get("confidence", 0.5)

            # Check for shared direction
            is_shared, direction = self._is_shared_direction(score_a, score_b)

            if is_shared:
                # Calculate trait similarity (0-1)
                diff = abs(score_a - score_b)
                trait_similarity = max(0, 1 - (diff / 10.0))

                # Apply confidence and weight
                avg_confidence = (conf_a + conf_b) / 2
                weight = self.SIN_WEIGHTS[sin]
                contribution = trait_similarity * avg_confidence * weight

                total_contribution += contribution

                breakdown.append({
                    "sin": sin,
                    "direction": direction,
                    "score_a": round(score_a, 2),
                    "score_b": round(score_b, 2),
                    "trait_similarity": round(trait_similarity, 3),
                    "avg_confidence": round(avg_confidence, 3),
                    "weight": weight,
                    "contribution": round(contribution, 4)
                })

        # Normalise to 0-1
        raw_score = total_contribution / self.TOTAL_WEIGHT if self.TOTAL_WEIGHT > 0 else 0

        return raw_score, breakdown

    def _is_shared_direction(
        self,
        score_a: float,
        score_b: float
    ) -> tuple[bool, Optional[str]]:
        """Determine if two scores share a direction (both virtue or both vice)."""

        # Check if either is in neutral zone
        if abs(score_a) <= self.NEUTRAL_THRESHOLD or abs(score_b) <=
self.NEUTRAL_THRESHOLD:
            return False, None

        both_vice = score_a > self.NEUTRAL_THRESHOLD and score_b >
self.NEUTRAL_THRESHOLD
        both_virtue = score_a < -self.NEUTRAL_THRESHOLD and score_b <
-self.NEUTRAL_THRESHOLD

        if both_vice:
            return True, "vice"
        elif both_virtue:
            return True, "virtue"
        else:
            return False, None

    def _get_quality_multiplier(self, quality_a: float, quality_b: float) -> float:
        """Get similarity multiplier based on profile qualities."""

        def tier(q):
            if q >= 80:
                return "high"
            elif q >= 60:
                return "moderate"
            else:
                return "low"

        tier_a, tier_b = tier(quality_a), tier(quality_b)

        multipliers = {
            ("high", "high"): 1.0,
            ("high", "moderate"): 0.9,
            ("moderate", "high"): 0.9,
            ("moderate", "moderate"): 0.8,
            ("high", "low"): 0.7,
            ("low", "high"): 0.7,
            ("moderate", "low"): 0.6,
            ("low", "moderate"): 0.6,
            ("low", "low"): 0.5
        }

        return multipliers.get((tier_a, tier_b), 0.5)

    def _combined_quality_tier(self, quality_a: float, quality_b: float) -> str:
        """Determine combined quality tier for display."""
        avg = (quality_a + quality_b) / 2
        if avg >= 80:
            return "high"
        elif avg >= 60:
            return "moderate"
        else:
            return "low"

    def _evaluate_threshold(self, score: float) -> tuple[str, str]:
        """Evaluate score against threshold and return tier + display mode."""

        t = self.stage2_threshold

        if score >= t + 0.20:
            return "strong_fit", "highlight"
        elif score >= t:
            return "good_fit", "standard"
        elif score >= t - 0.15:
            return "moderate_fit", "minimal"
        else:
            return "low_fit", "chemistry_focus"

    def _insufficient_data_result(self, profile_a: dict, profile_b: dict) -> dict:
        """Return result when one or both profiles have insufficient data."""
        return {
            "raw_score": 0.0,
            "adjusted_score": 0.0,
            "quality_multiplier": 0.0,
            "breakdown": [],
            "overlap_count": 0,
            "tier": "insufficient_data",
            "display_mode": "minimal",
            "quality_tier": "rejected"
        }

    def generate_explanation(self, similarity_result: dict) -> dict:
        """
        Generate user-facing explanation from similarity result.

        Returns:
            {
                "headline": str,
                "shared_traits": list[str],
                "summary": str
            }
        """
        breakdown = similarity_result["breakdown"]
        score = similarity_result["adjusted_score"]
        display_mode = similarity_result["display_mode"]

        # Handle no overlap
        if not breakdown:
            return {
                "headline": "Different perspectives, potential spark",
                "shared_traits": [],
                "summary": "You might balance each other out"
            }

        # Sort by contribution
        sorted_breakdown = sorted(breakdown, key=lambda x: x["contribution"], reverse=True)

        # Determine how many traits to show
        max_traits = {"highlight": 4, "standard": 3, "minimal": 2, "chemistry_focus": 1}
        show_count = max_traits.get(display_mode, 2)

        # Build descriptions
        shared_traits = []
        for item in sorted_breakdown[:show_count]:
            sin = item["sin"]
            direction = item["direction"]
            desc = self.TRAIT_DESCRIPTIONS[sin][direction]
            shared_traits.append(f"You're both {desc}")

        # Generate headline
        if score >= 0.60:
```

```
    headline = "Strong personality match!"
elif score >= 0.45:
    headline = "You have a lot in common"
elif score >= 0.30:
    headline = "Some shared traits"
else:
    headline = "Different perspectives, potential spark"

# Generate summary
overlap_count = len(breakdown)
if overlap_count >= 5:
    summary = f"You align on {overlap_count} personality traits"
elif overlap_count >= 3:
    summary = f"You share {overlap_count} key traits"
elif overlap_count >= 1:
    summary = "You have some things in common"
else:
    summary = "Your chemistry might surprise you"

return {
    "headline": headline,
    "shared_traits": shared_traits,
    "summary": summary,
    "overlap_count": overlap_count,
    "total_traits": 7
}
```

---

## 6.12 Summary

**The Sequential Matching Architecture:**

| Stage | Signal | Gate Type | Outcome |
|---|---|---|---|
| 1. Visual | Mutual swipes | Hard gate | Both must swipe right to proceed |
| 2. Personality | Perceived similarity | Soft gate | Score affects presentation, not permission |
| 3. Genetics | HLA compatibility | No gate | Informational display only |

**Perceived Similarity Calculation:**

| Component | Formula/Logic |
|---|---|
| Shared direction | Both scores > +0.5 OR both < -0.5 |
| Trait similarity | 1 - (\|score_a - score_b\| / 10) |
| Contribution | trait_similarity × avg_confidence × sin_weight |
| Raw score | Σ(contributions) / 7.4 |
| Adjusted score | raw_score × quality_multiplier |

**What This Achieves:**

1. **Reciprocity by construction** — Only mutual matches see personality data
2. **Positive framing only** — Users see what they share, never differences
3. **Confidence-aware** — Low-quality profiles contribute less to similarity
4. **Weighted by importance** — Wrath (conflict) matters most; Envy least
5. **Flexible thresholds** — Per-segment optimisation without code changes

**What It Doesn't Achieve:**

1. **Predictive validity** — Perceived similarity correlates with initial attraction, not long-term compatibility
2. **Actual similarity** — The algorithm explicitly ignores actual profile distance
3. **Negative screening** — Differences are never surfaced or penalised

**The Honest Framing:**

Harmonia's personality matching creates *perceived* connection, not *predicted* compatibility. Two users who share 4 of 7 traits will *feel* similar when those traits are highlighted — regardless of whether their differences would cause friction. This is by design: research shows perceived similarity drives initial attraction, while actual similarity does not predict relationship outcomes.

Understood. Let me rewrite Section 7 focused on personality integration only, referencing but not implementing the visual and HLA components.

---

## Section 7: Integration with Harmonia Matching

---

### 7.1 The Three Signals: What Each Component Measures

Harmonia's matching system combines three distinct signals, each measuring a different aspect of potential romantic connection:

| Signal | Weight | What It Measures | Implementation |
|---|---|---|---|
| **Visual** | 60% | "Will you find them physically attractive?" | MetaFBP system (separate document) |
| **Personality** | 30% | "Will you feel connected and similar?" | This report (Sections 1-7) |
| **Genetics** | 10% | "Is there biological chemistry potential?" | HLA system (separate document) |

**What Each Signal Is NOT:**

| Signal | Common Misconception | Reality |
|---|---|---|
| Visual | "Objective attractiveness" | Highly personal; A's rating of B ≠ C's rating of B |
| Personality | "Actual compatibility" | Perceived similarity matters; actual similarity does not predict attraction (β=0.006) |
| Genetics | "Guaranteed chemistry" | Emerging science with modest effect sizes; informational, not predictive |

**Signal Characteristics:**

| Property | Visual | Personality | Genetics |
|---|---|---|---|

| | | | |
|---|---|---|---|
| Symmetric? | No — A→B ≠ B→A | Yes — same score both directions | Yes — HLA dissimilarity is objective |
| Requires user data? | Photos + swipe history | Questionnaire responses | DNA upload (optional) |
| Cold start handling | Universal model | Cannot calculate | Cannot calculate |
| Personalises over time? | Yes (5-10 swipes) | No (fixed after assessment) | No (fixed after upload) |

## 7.2 The Visual Component: MetaFBP Reference

The visual signal represents 60% of the matching decision. It is implemented separately using MetaFBP (Meta-Learning for Facial Beauty Prediction).

**Key Characteristics (for integration purposes):**

| Property | Value |
|---|---|
| Rating scale | 1-5 (user rates sample photos) |
| Cold start accuracy | ~65% (universal model) |
| Personalised accuracy | ~78-80% after 5-10 ratings |
| Output format | Score 0-1, confidence 0-1, model_type |
| Symmetry | **Asymmetric** — visual_A→B ≠ visual_B→A |

**Interface Expected by MatchingService:**

```
# Visual service returns:
{
    "score": float,       # 0-1 attractiveness prediction
    "confidence": float,  # Based on personalisation level
    "model_type": str     # "universal" or "personalised"
}
```

Reference: Lin et al. (2023) "MetaFBP: Learning to Learn High-Order Predictor for Personalized Facial Beauty Prediction", ACM Multimedia. Production validation: Iris Dating (2M+ users).

## 7.3 The Personality Component: Perceived Similarity

The personality signal contributes 30% of the matching decision, based on the robust finding that **perceived similarity, not actual similarity, predicts initial attraction**.

**Research Foundation:**

Tidwell, Eastwick & Finkel (2013) conducted speed-dating studies measuring both actual trait similarity and perceived similarity:

| Predictor | Beta Coefficient | Significance |
|---|---|---|
| Actual similarity (trait-specific) | $\beta = 0.006$ | Not significant |
| Perceived similarity (trait-specific) | $\beta = 0.05$ | $p < 0.05$ (weak) |
| Perceived similarity (general) | $\beta = 0.75$ | $p < 0.001$ (very strong) |

**Key Finding:** General perceived similarity was **15× stronger** than trait-specific perceived similarity in predicting attraction.

This has been replicated in the 2024 meta-analysis covering 313 studies (From et al.), which confirmed that perceived similarity has stronger effects in early dating, while actual similarity becomes more important in established relationships.

**Mechanism:**

People infer perceived similarity via schemas without concrete evidence. If attracted to someone, they assume similarity across many traits (projection). The "astrology effect" leverages this: highlighting any shared characteristic increases perceived similarity, regardless of whether the trait meaningfully predicts compatibility.

**How Harmonia Implements This:**

Sections 4-6 defined the complete personality pipeline:

1. **Section 4 (Scoring):** Gemini extracts sin scores from scenario responses
2. **Section 5 (Aggregation):** Six questions aggregated into profile with quality tiers
3. **Section 6 (Similarity):** Positive overlap algorithm identifies shared trait directions

The match card then highlights these shared traits in natural language, creating the perceived similarity that drives attraction.

**Interface for MatchingService:**

```
# Personality service returns:
{
   "score": float,           # 0-1 similarity (adjusted_score from Section 6)
   "overlap_count": int,      # Number of shared trait directions
   "display_traits": list,    # Natural language descriptions for match card
   "headline": str            # "You have a lot in common" etc.
}
```

---

## 7.4 The Genetic Component: HLA Reference

The genetic signal contributes 10% of the matching decision, representing potential biological chemistry based on HLA/MHC dissimilarity. This is implemented separately.

**Research Foundation:**

| Paper | Finding | Limitation |
|---|---|---|
| Wedekind et al. (1995) | Women prefer scent of MHC-dissimilar men | Small sample |
| Kromer et al. (2016) | HLA-B/C dissimilarity → partnership satisfaction | N=508 |
| Croy et al. (2020) | No significant relationship in 3,691 couples | Contradicts earlier work |

**Honest Assessment:**

The 10% weighting may exceed strict scientific justification. Perplexity research (2025) suggests 2-3% would be more defensible. However, Harmonia positions HLA as an informational signal with "chemistry potential" framing.

**Interface Expected by MatchingService:**

```
# HLA service returns:
{
   "score": float,          # 0-100 chemistry score (None if unavailable)
   "available": bool,        # Whether both users have HLA data
   "display": dict | None    # Icon, label, detail for match card
}
```

**Key Design Decision:** Low/absent HLA scores produce no display (no negative messaging), not a penalty.

---

### 7.5 The Sequential Architecture

Harmonia uses a cascaded pipeline rather than a weighted average formula. The 60/30/10 weights represent **decision importance at each stage**, not coefficients in a linear combination.

**Pipeline Flow:**

```
┌─────────────────────────────────────────────────────────────
─────────┐
│ STAGE 1: VISUAL GATE                              │
│ User A sees User B's photos → Swipes (1-5 implicit via swipe) │
│ User B sees User A's photos → Swipes                  │
│                                   │
│ Gate: MUTUAL SWIPE REQUIRED                          │
│ Decision weight: 60% (most pairs eliminated here)          │
└─────────────────────────────────────────────────────────────
─────────┘
                    ↓
          [Only if mutual interest]
                    ↓
┌─────────────────────────────────────────────────────────────
─────────┐
│ STAGE 2: PERSONALITY REVEAL                          │
│ Calculate perceived similarity (this report)           │
│ Display shared traits ("You're both...")              │
│                                   │
│ Gate: SOFT (affects presentation, not permission)          │
│ Decision weight: 30%                            │
└─────────────────────────────────────────────────────────────
─────────┘
                    ↓
          [Personality revealed to users]
                    ↓
┌─────────────────────────────────────────────────────────────
─────────┐
│ STAGE 3: GENETICS INFO                            │
│ Display HLA chemistry (if available)                │
│                                   │
│ Gate: NONE — Informational only                      │
│ Decision weight: 10%                            │
```

```
└──────────────────────────────────────────────────────────────────
──────────┘
                    ↓
          RECIPROCAL SCORE CALCULATED
            (for ranking purposes)
```

**Why Sequential Beats Weighted Average:**

| Dimension | Weighted Average | Sequential Cascade |
|---|---|---|
| Computational cost | Calculate all signals for all pairs | Only calculate personality for mutual matches |
| Reciprocity | Cannot enforce | Built into Stage 1 (mutual swipe) |
| User psychology | Unnatural | Matches how humans evaluate: photos → personality → chemistry |
| Graceful degradation | Missing signal breaks formula | Missing genetics = skip display, no penalty |

---

## 7.6 Reciprocal Matching: Final Stage Application

Research consensus (Perplexity report, 2025) confirms that reciprocity should be applied **at the final ranking stage**, not per-stage. The formula:

reciprocal_score = √(score_A→B × score_B→A)

**Why Geometric Mean:**

The geometric mean penalises asymmetric attraction more heavily than arithmetic mean:

| Scenario | A→B | B→A | Arithmetic Mean | Geometric Mean |
|---|---|---|---|---|
| Both interested | 0.80 | 0.90 | 0.85 | 0.85 |
| One-sided | 0.90 | 0.10 | 0.50 | 0.30 |
| Both moderate | 0.60 | 0.60 | 0.60 | 0.60 |

**Combined Score Formula:**

For each direction (A→B and B→A):

```
combined_A_to_B = (0.60 × visual_A→B) + (0.30 × personality) + (0.10 × genetics)
combined_B_to_A = (0.60 × visual_B→A) + (0.30 × personality) + (0.10 × genetics)

reciprocal_score = √(combined_A_to_B × combined_B_to_A)
```

Note: Personality and genetics are symmetric (same value both directions), so asymmetry in the final reciprocal score comes entirely from the visual component.

---

## 7.7 Signal Asymmetry and Directional Scoring

**Signal Symmetry Matrix:**

| Signal | Calculation | Same Result Both Directions? |
|---|---|---|
| Visual | MetaFBP(viewer's prefs, target's photos) | **NO** — fully asymmetric |
| Personality | PerceivedSimilarity(A, B) | **YES** — symmetric |
| Genetics | HLADissimilarity(A, B) | **YES** — symmetric |

**Practical Implication:**

When calculating a match, you need:

- **Two** visual scores (A→B and B→A)
- **One** personality score (A↔B)
- **One** genetics score (A↔B)

**Weight Adjustment for Missing Signals:**

When signals are unavailable, redistribute weight proportionally:

| Available Signals | Adjusted Weights |
|---|---|
| All three | 60% / 30% / 10% (unchanged) |
| Visual + Personality (no genetics) | 67% / 33% / 0% |

| | |
|---|---|
| Visual + Genetics (no personality) | 86% / 0% / 14% |
| Visual only | 100% / 0% / 0% |

---

## 7.8 Personality's Role in the Matching Service

The MatchingService orchestrates all three signals. This section specifies how personality integrates with the other components.

**Personality Integration Points:**

```
class MatchingService:
    """
    Orchestrates three-signal matching pipeline.

    Visual and HLA services are injected dependencies.
    Personality service uses SimilarityService from Section 6.
    """

    DEFAULT_WEIGHTS = {
        "visual": 0.60,
        "personality": 0.30,
        "genetics": 0.10
    }

    def __init__(
        self,
        visual_service,          # External: MetaFBP implementation
        personality_service,     # This report: SimilarityService wrapper
        hla_service,             # External: HLA implementation
        weights: dict = None
    ):
        self.visual = visual_service
        self.personality = personality_service
        self.hla = hla_service
        self.weights = weights or self.DEFAULT_WEIGHTS
```

**Personality Calculation Within Match Flow:**

```
def _calculate_personality(self, user_a: dict, user_b: dict) -> dict:
    """
    Calculate personality similarity with availability check.
    Uses SimilarityService from Section 6.
    """
    profile_a = user_a.get("personality_profile")
    profile_b = user_b.get("personality_profile")

    # Handle missing profiles
    if profile_a is None or profile_b is None:
        return {
            "score": 0.0,
            "available": False,
            "reason": "missing_profile"
        }

    # Handle rejected profiles (from Section 5 quality control)
    if profile_a.get("quality", {}).get("tier") == "rejected" or \
       profile_b.get("quality", {}).get("tier") == "rejected":
        return {
            "score": 0.0,
            "available": False,
            "reason": "rejected_profile"
        }

    # Calculate using SimilarityService (Section 6)
    result = self.personality.compute_similarity(profile_a, profile_b)

    return {
        "score": result["score"],           # 0-1 adjusted similarity
        "available": True,
        "overlap_count": result["overlap_count"],
        "display_traits": result["display_traits"],
        "headline": result["headline"]
    }
```

**Combining Personality Into Final Score:**

```
def _combine_scores(
    self,
    visual_score: float,
    personality_score: float,
    genetics_score: float,
    weights: dict
) -> float:
    """
    Combine signals into single directional score.

    Personality contributes 30% (default) of combined score.
    """
    combined = 0.0

    if "visual" in weights:
        combined += weights["visual"] * visual_score
    if "personality" in weights:
        combined += weights["personality"] * personality_score
    if "genetics" in weights:
        combined += weights["genetics"] * genetics_score

    return combined
```

**Match Card Assembly (Personality Section):**

```
def _assemble_match_card(
    self,
    user_a: dict,
    user_b: dict,
    personality: dict,
    genetics: dict,
    reciprocal_score: float
) -> dict:
    """
    Assemble display data for match card.

    Personality section shows shared traits (the "astrology effect").
    """
    card = {
        "match_id": f"{user_a['user_id']}_{user_b['user_id']}",
        "score_display": f"{int(reciprocal_score)}%"
    }

    # Personality section — the perceived similarity display
    if personality["available"]:
        card["personality"] = {
            "headline": personality["headline"],
            "shared_traits": personality["display_traits"][:3],  # Max 3 shown
            "overlap_count": personality["overlap_count"]
        }

    # Chemistry section (from HLA service)
    if genetics.get("available") and genetics.get("display"):
        card["chemistry"] = genetics["display"]

    return card
```

---

## 7.9 Configuration and Tuning

**Default Configuration:**

```
MATCHING_CONFIG = {
    "weights": {
        "visual": 0.60,
        "personality": 0.30,
        "genetics": 0.10
    },

    "thresholds": {
        "personality_minimum": 0.25,  # Below this, flag as "low fit"
        "symmetry_mutual": 0.70,      # Above this, consider "mutual"
    },

    "display": {
        "max_shared_traits": 3,
        "show_genetics_above": 25     # Only show chemistry if score >= 25
    },

    "relationship_goal_adjustments": {
        "casual": {
            "visual": 0.70,
            "personality": 0.25,
            "genetics": 0.05
        },
        "serious": {
            "visual": 0.50,
            "personality": 0.40,
            "genetics": 0.10
        }
    }
}
```

**Personality-Specific Thresholds:**

From Section 6.9, personality similarity produces tier assignments:

| Score Range | Tier | Display Mode | Traits Shown |
|---|---|---|---|
| ≥0.60 | strong_fit | highlight | 4 traits |
| ≥0.40 | good_fit | standard | 3 traits |
| ≥0.25 | moderate_fit | minimal | 2 traits |
| <0.25 | low_fit | chemistry_focus | 1 trait |

These tiers affect how prominently personality appears on the match card, but do not block matches.

**Recommended A/B Tests:**

| Test | Control | Treatment | Metric |
|---|---|---|---|
| Personality weight | 30% | 40% | Message response rate |
| Trait display count | 3 | 5 | Perceived connection (survey) |
| Headline style | Generic | Specific count | Engagement rate |

---

## 7.10 Summary

**Personality's Contribution to Matching:**

| Aspect | How Personality Contributes |
|---|---|
| Match score | 30% of combined directional score |
| Match card | Shared traits display ("You're both...") |
| User perception | Creates "astrology effect" — sense of connection |
| Threshold control | Soft gate affects presentation, not permission |

**What Personality Matching Achieves:**

| Goal | Mechanism |
|---|---|
| Perceived connection | Highlighting shared trait directions |
| Conversation starters | Natural language trait descriptions |
| Warm introduction | "You have a lot in common" framing |

**What It Doesn't Achieve:**

| Limitation | Honest Framing |
|---|---|
| Actual compatibility prediction | Perceived ≠ actual; β=0.006 for actual similarity |
| Long-term success prediction | Initial attraction signal only |
| Complementarity matching | System finds similarity, not opposites |

**Integration Summary:**

User completes 6 scenarios
↓
Gemini extracts sin scores (Section 4)
↓
Aggregation produces profile (Section 5)
↓
After mutual swipe...
↓
SimilarityService calculates overlap (Section 6)
↓
MatchingService combines with visual + genetics
↓
Match card displays shared traits
↓
Users perceive similarity → attraction enhanced

**Research-Backed Confidence:**

| Component | Research Support | Confidence |
|---|---|---|
| Perceived > actual similarity | β=0.75 vs β=0.006, replicated extensively | **High** |

| | | |
|---|---|---|
| Positive overlap approach | Tidwell et al. mechanism | **High** |
| 30% weight allocation | No direct research; design decision | **Medium** |
| Sin-specific weighting | Theoretical (conflict > lifestyle) | **Low-Medium** |

---

End of Section 7.

## Section 8: Error Handling and Validation

This section documents the complete error handling strategy for Gemini API integration in Harmonia's personality analysis pipeline. Production reliability requires handling API failures, malformed responses, and graceful degradation when upstream services fail.

---

### 8.1 Error Taxonomy

Errors are categorised by source, retryability, and required action.

**Tier 1: API-Level Errors**

These originate from the Gemini API itself:

| HTTP Code | Error Type | Retryable | Description | Action |
|---|---|---|---|---|
| 429 | RESOURCE_EXHAUSTED | ✅ Yes | Rate limit exceeded (RPM/TPM/RPD) | Exponential backoff; respect `retry-after` header |
| 500 | INTERNAL | ✅ Yes | Server error; possibly input too long | Retry 2-3 times; reduce context if persistent |
| 503 | UNAVAILABLE | ✅ Yes | Model overloaded or maintenance | Backoff then fallback to secondary model |
| 504 | DEADLINE_EXCEEDED | ⚠️ Maybe | Request exceeded server timeout | Increase client timeout; simplify prompt |

| | | | | |
|---|---|---|---|---|
| 400 | INVALID_ARGUMENT | ❌ No | Malformed request or invalid parameters | Fix request immediately; log for debugging |
| 403 | PERMISSION_DENIED | ❌ No | Invalid API key or insufficient permissions | Verify credentials; alert ops team |
| 404 | NOT_FOUND | ❌ No | Referenced resource not found | Validate file paths; check model name |

**Tier 2: Response-Level Errors**

API returns 200 OK, but response content is problematic:

| Issue | Detection | Frequency | Action |
|---|---|---|---|
| Empty response | `len(candidates[0].content.parts) == 0` | <1% | Check `finish_reason`; retry once |
| Truncated JSON | `finish_reason == "MAX_TOKENS"` | <2% | Increase `max_output_tokens`; retry |
| Safety block | `finish_reason == "SAFETY"` | <0.5% | Log blocked content; return graceful failure |
| Recitation block | `finish_reason == "RECITATION"` | <0.1% | Rephrase prompt; retry once |

| | | | |
|---|---|---|---|
| Malformed JSON | `json.JSONDecodeError` | 1-5% | Apply JSON extraction pipeline |

**Tier 3: Validation Errors**

JSON parses successfully, but content fails business rules:

| Issue | Detection | Action |
|---|---|---|
| Score out of range | `score < -5 or score > 5` | Clamp to [-5, +5]; flag for review |
| Confidence out of range | `confidence < 0 or confidence > 1` | Clamp to [0, 1] |
| Missing sin | `sin not in response["sins"]` | Use default (score=0, confidence=0.3) |
| Wrong sin name | `sin not in VALID_SINS` | Attempt fuzzy match; else skip |
| Missing evidence | `evidence is None or len(evidence) < 5` | Use placeholder; reduce confidence by 0.2 |

**Tier 4: Logic Errors**

Internal application errors:

| Issue | Detection | Action |
|---|---|---|
| Profile not found | Database lookup fails | Return 404 to client |

| | | |
|---|---|---|
| Duplicate processing | Same Q&A processed twice | Idempotency check; skip duplicate |
| Stale data | Profile version mismatch | Fetch fresh profile; retry calculation |

---

## 8.2 Retry Strategy

**Core Parameters:**

| Parameter | Value | Rationale |
|---|---|---|
| Initial delay | 1 second | Google's recommended starting point |
| Max delay | 60 seconds | Prevents excessive wait times |
| Backoff multiplier | 2× (exponential) | Standard exponential backoff |
| Jitter | 0-1 second random | Prevents thundering herd |
| Max retries | 5 attempts | Balances reliability vs latency |
| Total deadline | 300 seconds | Hard cap on retry window |

**Retry Sequence Example:**

```
Attempt 1: Immediate
Attempt 2: 1s + jitter (~1.3s)
Attempt 3: 2s + jitter (~2.7s)
Attempt 4: 4s + jitter (~4.5s)
Attempt 5: 8s + jitter (~8.2s)
Attempt 6: Would be 16s, but max_retries=5 exceeded → fail
```

**Model Fallback Chain:**

Based on production benchmarks, Flash outperforms Pro for structured JSON extraction:

| Metric | Gemini 3.0 Flash | Gemini 3.0 Pro |
|---|---|---|
| Structured output success | 95-99% | 95-98% |
| Latency (2K output) | ~9 seconds | ~27 seconds |
| Cost per 1M tokens (input) | $0.50 | $2.00 |
| Availability | >99.7% | 99.4-99.6% |

**Fallback Strategy:**

```
Primary: Gemini 3.0 Flash (30s timeout, 3 retries)
  ↓ All retries exhausted
Fallback: Gemini 3.0 Pro (60s timeout, 2 retries)
  ↓ All retries exhausted
Final: Return graceful failure
```

**Error-Specific Retry Rules:**

python

```python
RETRY_CONFIG = {
    429: {"retry": True, "max_attempts": 5, "respect_retry_after": True},
    500: {"retry": True, "max_attempts": 3, "note": "Reduce context if persistent"},
    503: {"retry": True, "max_attempts": 5, "fallback_model": True},
    504: {"retry": True, "max_attempts": 2, "increase_timeout": True},
    400: {"retry": False, "action": "fix_request"},
    403: {"retry": False, "action": "check_credentials"},
    404: {"retry": False, "action": "validate_resource"},
}
```

---

## 8.3 Response Validation

**Step 1: Check HTTP Status**

python

```python
if response.status_code != 200:
    error_code = response.status_code
    if RETRY_CONFIG[error_code]["retry"]:
        raise RetryableError(error_code, response.json())
    else:
        raise PermanentError(error_code, response.json())
```

**Step 2: Check Finish Reason**

The `finish_reason` field indicates whether generation completed successfully:

| finish_reason | JSON Valid? | Action |
|---|---|---|
| STOP | ✅ Complete | Parse normally |
| MAX_TOKENS | ❌ Truncated | Retry with higher token limit |
| SAFETY | ❌ Blocked | Log; return graceful failure |
| RECITATION | ❌ Blocked | Rephrase prompt; retry once |
| SPII | ❌ Blocked | Cannot proceed; return failure |
| OTHER | ❌ Unknown | Log full response; investigate |

python

```python
FINISH_REASON_HANDLERS = {
    "STOP": lambda r: parse_json(r),
    "MAX_TOKENS": lambda r: raise_truncation_error(r),
    "SAFETY": lambda r: handle_safety_block(r),
    "RECITATION": lambda r: raise_recitation_error(r),
}

finish_reason = response.candidates[0].finish_reason
handler = FINISH_REASON_HANDLERS.get(finish_reason, handle_unknown)
result = handler(response)
```

**Step 3: JSON Extraction Pipeline**

Even with `response_json_schema` enforcement, 1-5% of responses require cleaning:

python

```python
def extract_json(response_text: str) -> dict:
    """
    Extract JSON from Gemini response with fallback handling.

    Success rates by stage:
    - Stage 1 (direct parse): ~94%
    - Stage 2 (markdown extraction): ~4%
    - Stage 3 (prefix removal): ~1%
    - Stage 4 (jsonrepair): ~1%
    """

    # Stage 1: Direct parse (happy path)
    try:
        return json.loads(response_text)
    except json.JSONDecodeError:
        pass

    # Stage 2: Remove markdown code blocks
    # Pattern: ```json\n{...}\n``` or ```\n{...}\n```
    match = re.match(r'```(?:json)?\n([\s\S]*?)\n```', response_text.strip())
    if match:
```

```
    json_text = match.group(1).strip()
else:
    json_text = response_text.strip()

# Stage 3: Remove common prefixes
prefixes = [
    r"^Here'?s? the JSON:\s*",
    r"^Here is the JSON:\s*",
    r"^JSON response:\s*",
    r"^Response:\s*",
    r"^Output:\s*",
]
for prefix in prefixes:
    json_text = re.sub(prefix, '', json_text, flags=re.IGNORECASE)

# Try parsing cleaned text
try:
    return json.loads(json_text)
except json.JSONDecodeError:
    pass

# Stage 4: Attempt repair (last resort)
try:
    from jsonrepair import repair_json
```

```
        repaired = repair_json(json_text)
        return json.loads(repaired)
    except Exception as e:
        raise JSONExtractionError(
            f"Failed to parse JSON after all stages. "
            f"Sample: {json_text[:200]}"
        )
```

**Step 4: Schema Validation**

After JSON extraction, validate against expected structure:

python

```
from pydantic import BaseModel, Field, validator
from typing import Dict, List


class SinScore(BaseModel):
    score: float = Field(ge=-5, le=5)
    confidence: float = Field(ge=0, le=1)
    evidence: str = Field(min_length=1)

    @validator('score')
    def clamp_score(cls, v):
        return max(-5, min(5, v))

    @validator('confidence')
    def clamp_confidence(cls, v):
        return max(0, min(1, v))


class GeminiResponse(BaseModel):
    sins: Dict[str, SinScore]
    style: Dict[str, str] = None

    @validator('sins')
    def validate_sins(cls, v):
        required = {"greed", "pride", "lust", "wrath", "gluttony", "envy", "sloth"}
        missing = required - set(v.keys())
        if missing:
            # Fill missing with defaults
            for sin in missing:
                v[sin] = SinScore(score=0, confidence=0.3, evidence="[No signal detected]")
        return v


def validate_response(json_data: dict) -> GeminiResponse:
    """Validate and normalise Gemini response."""
    try:
        return GeminiResponse(**json_data)
    except ValidationError as e:
        raise SchemaValidationError(f"Response doesn't match schema: {e}")
```

---

### 8.4 Default Fallback Values

When partial data is available, use sensible defaults rather than failing entirely.

**Sin-Level Defaults:**

| Scenario | Default Score | Default Confidence | Rationale |
|---|---|---|---|
| Missing sin in response | 0 | 0.30 | Neutral with low confidence |
| Score out of range | Clamped to [-5, +5] | Original | Preserve confidence |
| Confidence out of range | Original | Clamped to [0, 1] | Preserve score |
| Empty evidence | Original | Reduced by 0.20 | Penalise missing justification |
| Parsing failed (single sin) | 0 | 0.25 | Very low confidence |

**Question-Level Defaults:**

| Scenario | Action | Impact |
|---|---|---|
| Single question fails | Skip question; use remaining 5 | Reduced profile confidence |
| 2 questions fail | Proceed with 4 questions | Minimum viable |

| | | |
|---|---|---|
| 3+ questions fail | Request re-assessment | Cannot generate reliable profile |

**Profile-Level Defaults:**

| Scenario | Action |
|---|---|
| <4 questions answered | Return `profile_status: "incomplete"` |
| Average confidence <0.50 | Return `quality_tier: "low"` |
| Gemini fully unavailable | Return `profile_status: "pending"` with retry queue |

**When to Fail vs Default:**

python

```python
FAILURE_CONDITIONS = {
    # Hard failures - cannot proceed
    "api_key_invalid": True,
    "account_suspended": True,
    "all_retries_exhausted": True,
    "safety_block_on_question": False,  # Skip question, not hard fail

    # Soft failures - use defaults
    "single_sin_missing": False,
    "confidence_slightly_low": False,
    "evidence_empty": False,
}
```

---

### 8.5 Graceful Degradation

**Single Question Failure:**

python

```python
def process_question_batch(questions: List[dict]) -> dict:
    """
    Process 6 questions with graceful degradation.

    Degradation tiers:
    - 6 successful: Full quality profile
    - 5 successful: Good quality (one question dropped)
    - 4 successful: Minimum viable profile
    - <4 successful: Request re-assessment
    """
    results = []
    failures = []

    for q in questions:
        try:
            result = call_gemini_with_retry(q)
            results.append(result)
        except GeminiAPIError as e:
            failures.append({"question_id": q["id"], "error": str(e)})
            logger.warning(f"Question {q['id']} failed: {e}")

    # Determine degradation tier
    success_count = len(results)

    if success_count >= 6:
        quality_tier = "full"
    elif success_count >= 5:
        quality_tier = "good"
    elif success_count >= 4:
        quality_tier = "minimum"
    else:
        return {
            "status": "insufficient_data",
            "successful_questions": success_count,
            "failures": failures,
            "action": "request_reassessment"
        }

    return {
        "status": "success",
        "quality_tier": quality_tier,
        "results": results,
        "failures": failures
    }
```

**Total API Failure:**

When both Flash and Pro fail after all retries:

python

```
def handle_total_failure(user_id: str, questions: List[dict]) -> dict:
    """
    Handle complete Gemini API unavailability.

    Strategy:
    1. Queue for background retry
    2. Return placeholder profile
    3. Notify user of delay
    """

    # Queue for retry (exponential backoff at queue level)
    retry_job = {
        "user_id": user_id,
        "questions": questions,
        "attempt": 1,
        "next_retry": datetime.utcnow() + timedelta(minutes=5),
        "max_attempts": 10
    }
    queue_personality_job(retry_job)


    # Return placeholder
    return {
        "status": "pending",
        "profile": None,
        "message": "Your personality profile is being processed. Check back in a few minutes.",
        "estimated_completion": datetime.utcnow() + timedelta(minutes=10),
        "job_id": retry_job["id"]
    }
```

**Matching Without Personality:**

If a user's personality profile is unavailable during matching:

python

```
def calculate_match_score(user_a: dict, user_b: dict) -> dict:
    """
    Calculate match score with graceful degradation.


    Weight reallocation when personality unavailable:
    - Normal: 60% visual, 30% personality, 10% HLA
    - No personality: 85% visual, 15% HLA
    - No HLA: 65% visual, 35% personality
    - Neither: 100% visual
    """
```

```
has_personality = user_a.get("personality") and user_b.get("personality")
has_hla = user_a.get("hla") and user_b.get("hla")

if has_personality and has_hla:
    weights = {"visual": 0.60, "personality": 0.30, "hla": 0.10}
elif has_personality and not has_hla:
    weights = {"visual": 0.65, "personality": 0.35, "hla": 0.0}
elif not has_personality and has_hla:
    weights = {"visual": 0.85, "personality": 0.0, "hla": 0.15}
else:
    weights = {"visual": 1.0, "personality": 0.0, "hla": 0.0}

# Calculate weighted score...
```

---

## 8.6 Logging and Observability

**What to Log:**

| Event | Level | Fields | Purpose |
|---|---|---|---|
| API call start | INFO | model, user_id, question_id, prompt_tokens | Request tracking |
| API call success | INFO | model, latency_ms, output_tokens, finish_reason | Performance monitoring |

| | | | |
|---|---|---|---|
| API call failure | WARNING | model, error_code, error_message, attempt_number | Error tracking |
| Retry attempt | WARNING | model, attempt, wait_time, error_type | Retry analysis |
| Model fallback | WARNING | from_model, to_model, reason | Fallback frequency |
| JSON extraction stage | DEBUG | stage_name, success, sample | Parsing analysis |
| Validation failure | WARNING | field, expected, actual, action_taken | Data quality |
| Total failure | ERROR | user_id, all_errors, queued_for_retry | Critical alerts |

**Structured Logging Format:**

python

```python
import structlog


logger = structlog.get_logger()


def log_api_call(
    event: str,
    model: str,
    user_id: str,
    **kwargs
):
    """Structured logging for Gemini API calls."""
    logger.info(
        event,
        service="personality_analysis",
        model=model,
        user_id=user_id,
        timestamp=datetime.utcnow().isoformat(),
        **kwargs
    )

# Usage
log_api_call(
    event="gemini_api_success",
    model="gemini-3-flash-preview",
    user_id="user_123",
    latency_ms=2340,
    output_tokens=1847,
    finish_reason="STOP",
    question_id="q_456"
)
```

**Alerting Thresholds:**

| Metric | Warning | Critical | Action |
|---|---|---|---|

| | | | |
|---|---|---|---|
| Error rate (5-min window) | >2% | >5% | Page on-call |
| Fallback rate | >10% | >25% | Investigate Flash availability |
| P95 latency | >30s | >60s | Check model load |
| JSON extraction failures | >5% | >10% | Review prompt template |
| Total failures (hourly) | >5 | >20 | Check API status; consider Vertex AI |

**Dashboard Metrics:**

python

```
# Key metrics to track
METRICS = {
    "gemini_api_calls_total": Counter("Total API calls", ["model", "status"]),
    "gemini_api_latency": Histogram("API latency", ["model"]),
    "gemini_retry_count": Counter("Retry attempts", ["model", "error_code"]),
    "gemini_fallback_count": Counter("Model fallbacks", ["from_model", "to_model"]),
    "json_extraction_stage": Counter("JSON extraction by stage", ["stage"]),
    "validation_errors": Counter("Validation errors", ["field", "error_type"]),
    "profile_quality_tier": Counter("Profile quality distribution", ["tier"]),
}
```

## 8.7 Implementation Patterns

**Complete GeminiService with Error Handling:**

python

```
"""
Production Gemini service for Harmonia personality analysis.

Features:
- Automatic retry with exponential backoff
- Model fallback (Flash → Pro)
- JSON extraction with multi-stage cleaning
- Schema validation with Pydantic
- Structured logging
- Graceful degradation
"""

import asyncio
import json
import re
import logging
from datetime import datetime
from enum import Enum
from typing import Dict, Any, Optional, List

from google import genai
from google.api_core.exceptions import (
    ResourceExhausted,
    ServiceUnavailable,
    InternalServerError,
    InvalidArgument,
    PermissionDenied
)
from pydantic import BaseModel, Field, validator, ValidationError
from tenacity import (
    retry,
    stop_after_attempt,
    wait_exponential_jitter,
    retry_if_exception_type
)

logger = logging.getLogger(__name__)


# ===== MODELS =====

class GeminiModel(Enum):
    FLASH = "gemini-3-flash-preview"
    PRO = "gemini-3-pro-preview"
```

```python
class SinScore(BaseModel):
    """Individual sin score from Gemini response."""
    score: float = Field(ge=-5, le=5)
    confidence: float = Field(ge=0, le=1)
    evidence: str = Field(min_length=1, default="[No evidence provided]")

    @validator('score', pre=True)
    def clamp_score(cls, v):
        if isinstance(v, (int, float)):
            return max(-5, min(5, float(v)))
        return v

    @validator('confidence', pre=True)
    def clamp_confidence(cls, v):
        if isinstance(v, (int, float)):
            return max(0, min(1, float(v)))
        return v



class GeminiParseResponse(BaseModel):
    """Validated Gemini response structure."""
    sins: Dict[str, SinScore]
    style: Optional[Dict[str, str]] = None
    @validator('sins', pre=True)
    def ensure_all_sins(cls, v):
        required = {"greed", "pride", "lust", "wrath", "gluttony", "envy", "sloth"}
        if isinstance(v, dict):
            missing = required - set(v.keys())
            for sin in missing:
                v[sin] = {"score": 0, "confidence": 0.3, "evidence": "[No signal detected]"}
        return v



# ===== EXCEPTIONS =====


class GeminiError(Exception):
    """Base exception for Gemini operations."""
    pass


class JSONExtractionError(GeminiError):
    """Failed to extract valid JSON from response."""
    pass


class SchemaValidationError(GeminiError):
    """Response doesn't match expected schema."""
    pass
```

```
class FallbackExhaustedError(GeminiError):
    """All fallback models failed."""
    pass

class SafetyBlockError(GeminiError):
    """Content blocked by safety filters."""
    pass


# ===== JSON EXTRACTION =====

def extract_json(response_text: str) -> Dict[str, Any]:
    """
    Extract JSON from Gemini response with multi-stage fallback.

    Stages:
    1. Direct parse (~94% success)
    2. Markdown extraction (~4%)
    3. Prefix removal (~1%)
    4. jsonrepair (~1%)
    """
    # Stage 1: Direct parse
    try:
        return json.loads(response_text)
```

```python
except json.JSONDecodeError:
    pass

# Stage 2: Markdown code blocks
match = re.match(r'```(?:json)?\n([\s\S]*?)\n```', response_text.strip())
json_text = match.group(1).strip() if match else response_text.strip()

# Stage 3: Common prefixes
prefixes = [
    r"^Here'?s? the JSON:\s*",
    r"^Here is the JSON:\s*",
    r"^JSON response:\s*",
    r"^Response:\s*",
]
for prefix in prefixes:
    json_text = re.sub(prefix, '', json_text, flags=re.IGNORECASE)

try:
    return json.loads(json_text)
except json.JSONDecodeError:
    pass

# Stage 4: Repair
try:
```

```
        from jsonrepair import repair_json
        repaired = repair_json(json_text)
        return json.loads(repaired)
    except Exception as e:
        raise JSONExtractionError(f"JSON extraction failed. Sample: {json_text[:200]}")


# ===== RETRY CONFIGURATION =====

@retry(
    wait=wait_exponential_jitter(initial=1, max=60, jitter=5),
    stop=stop_after_attempt(5),
    retry=retry_if_exception_type((
        ResourceExhausted,
        ServiceUnavailable,
        InternalServerError
    )),
    reraise=True
)
async def call_gemini_with_retry(
    prompt: str,
    model: GeminiModel,
    schema: dict,
    timeout: int,
    api_key: str
) -> str:
    """
    Call Gemini API with automatic retry on transient errors.

    Retries: 429, 503, 500
    No retry: 400, 403, 404
    """
    client = genai.Client(api_key=api_key)

    try:
        response = await asyncio.wait_for(
            client.models.generate_content_async(
                model=model.value,
                contents=prompt,
                config={
                    "response_mime_type": "application/json",
                    "response_json_schema": schema,
                    "max_output_tokens": 4096
                }
            ),
            timeout=timeout
        )
        # Check finish reason
        finish_reason = response.candidates[0].finish_reason

        if finish_reason == "MAX_TOKENS":
            raise GeminiError("Response truncated - hit token limit")
        elif finish_reason == "SAFETY":
            raise SafetyBlockError(f"Content blocked: {response.candidates[0].safety_ratings}")
        elif finish_reason == "RECITATION":
            raise GeminiError("Content blocked - recitation")
        elif finish_reason != "STOP":
            raise GeminiError(f"Unexpected finish_reason: {finish_reason}")

        return response.text

    except asyncio.TimeoutError:
        raise GeminiError(f"Timeout after {timeout}s")
    except (InvalidArgument, PermissionDenied) as e:
        raise GeminiError(f"Non-retryable error: {e}")


# ===== MAIN SERVICE =====

class GeminiService:
    """
```

```
Production Gemini service with fallback and graceful degradation.

Fallback chain: Flash (primary) → Pro (secondary)
"""

SCHEMA = GeminiParseResponse.schema()

def __init__(self, api_key: str):
    self.api_key = api_key
    self.primary_model = GeminiModel.FLASH
    self.fallback_model = GeminiModel.PRO
    self.primary_timeout = 30
    self.fallback_timeout = 60

async def parse_response(
    self,
    question: str,
    answer: str,
    user_id: str
) -> GeminiParseResponse:
    """
    Parse a single question-answer pair.

    Returns validated GeminiParseResponse.
```

```
Raises FallbackExhaustedError if all models fail.
"""
prompt = self._build_prompt(question, answer)

models = [
    (self.primary_model, self.primary_timeout, 3),
    (self.fallback_model, self.fallback_timeout, 2)
]

last_error = None

for model, timeout, max_retries in models:
    try:
        logger.info(
            "gemini_attempt",
            model=model.value,
            user_id=user_id
        )

        response_text = await call_gemini_with_retry(
            prompt=prompt,
            model=model,
            schema=self.SCHEMA,
            timeout=timeout,
            api_key=self.api_key
        )

        # Extract and validate
        json_data = extract_json(response_text)
        validated = GeminiParseResponse(**json_data)

        logger.info(
            "gemini_success",
            model=model.value,
            user_id=user_id
        )

        return validated

    except SafetyBlockError as e:
        # Don't fallback on safety blocks
        logger.warning("gemini_safety_block", user_id=user_id, error=str(e))
        raise

    except Exception as e:
        logger.warning(
            "gemini_failure",
            model=model.value,
                user_id=user_id,
                error=str(e)
            )
            last_error = e
            continue

        raise FallbackExhaustedError(f"All models failed. Last error: {last_error}")

    def _build_prompt(self, question: str, answer: str) -> str:
        """Build the parsing prompt."""
        return f"""
Role: Expert Psychological Profiler using the Seven Deadly Sins framework.
Task: Analyze this response for Seven Deadly Sins traits.

Question: "{question}"
Answer: "{answer}"

Score each sin from -5 (extreme virtue) to +5 (extreme vice).
Provide confidence (0-1) and quote evidence from the response.

Output valid JSON matching the schema.
"""

    async def process_question_batch(
        self,
        questions: List[dict],
        user_id: str
    ) -> dict:
        """
        Process multiple questions with graceful degradation.

        Returns profile even if some questions fail.
        """
        results = []
        failures = []

        for q in questions:
            try:
            result = await self.parse_response(
                question=q["question"],
                answer=q["answer"],
                user_id=user_id
            )
            results.append({
                "question_id": q["id"],
                "response": result.dict()
            })
            except SafetyBlockError:
```

```
        failures.append({
            "question_id": q["id"],
            "error": "safety_block",
            "recoverable": False
        })
    except FallbackExhaustedError as e:
        failures.append({
            "question_id": q["id"],
            "error": str(e),
            "recoverable": True
        })

# Determine quality tier
success_count = len(results)
if success_count >= 6:
    quality_tier = "full"
elif success_count >= 5:
    quality_tier = "good"
elif success_count >= 4:
    quality_tier = "minimum"
else:
    quality_tier = "insufficient"

return {
    "status": "success" if success_count >= 4 else "insufficient_data",
    "quality_tier": quality_tier,
    "successful_questions": success_count,
    "results": results,
    "failures": failures,
    "timestamp": datetime.utcnow().isoformat()
}
```

---

## 8.8 Summary

**Error Handling Principles:**

| Principle | Implementation |
|---|---|
| Retry transient failures | Exponential backoff for 429, 500, 503 |
| Fail fast on permanent errors | No retry for 400, 403, 404 |
| Fallback gracefully | Flash → Pro model chain |
| Use sensible defaults | Missing sins get score=0, confidence=0.3 |
| Degrade gracefully | 4+ questions = usable profile |
| Log everything | Structured logging with alerting thresholds |

**Key Thresholds:**

| Parameter | Value |
|---|---|
| Max retries per model | 5 (Flash), 2 (Pro) |
| Initial backoff | 1 second |
| Max backoff | 60 seconds |
| Flash timeout | 30 seconds |
| Pro timeout | 60 seconds |
| Minimum questions for profile | 4 |
| Minimum profile confidence | 0.50 |

**What This Section Achieves:**

1. Comprehensive error taxonomy for debugging
2. Research-backed retry parameters
3. Multi-stage JSON extraction (handles 99%+ of responses)
4. Pydantic validation with automatic clamping
5. Graceful degradation at question and profile levels
6. Production-ready logging and alerting
7. Complete service implementation

**Honest Limitations:**

- jsonrepair may hallucinate structure on truncated JSON
- Safety blocks cannot be bypassed; content is lost
- AI Studio has no SLA; consider Vertex AI for critical workloads

- Flash availability varies; monitor fallback rates

# Section 9: Deployment

This section documents the complete deployment configuration for Harmonia's personality analysis service on Railway, optimised for long-running Gemini API calls (10-60 seconds per request).

---

## 9.1 Railway Platform Overview

**Why Railway for Harmonia:**

Railway provides a managed platform that handles the infrastructure complexity of deploying AI-powered services. Key advantages for Harmonia:

| Capability | Benefit for Harmonia |
|---|---|
| No hardcoded request timeout | 60-second Gemini calls complete without interruption |
| Graceful shutdown with drain period | In-flight requests survive redeployments |
| JSON-indexed logging | Real-time Gemini latency tracking via `@gemini_latency_ms` |
| IPv6 private networking | 5ms latency between API and worker services |
| GitHub auto-deploy | Push to `main` triggers deployment automatically |

**Tier Comparison:**

| Feature | Starter ($5/mo base) | Pro ($20/mo base) |
|---|---|---|
| Included credits | $5 usage | $20 usage |
| App sleeping | Yes (after inactivity) | No (always on) |
| Cold start impact | 10-30 seconds | None |
| Custom domains | 2 per service | Unlimited |
| Monitoring alerts | ❌ | ✅ |

|  |  |  |
|---|---|---|
| Priority support | ❌ | ✅ |
| SLA | None | Available |

**Recommendation:**

- **Beta (20-100 testers):** Starter tier acceptable; users may experience occasional cold starts
- **Production (100+ users/day):** Pro tier required for always-on availability and monitoring alerts

---

## 9.2 Project Structure

```
harmonia-personality/
├── main.py               # FastAPI application
├── config.py             # Environment configuration
├── services/
│   ├── gemini_service.py   # Gemini API integration
│   ├── similarity_service.py
│   └── profile_service.py
├── railway.json          # Railway build/deploy config
├── Procfile              # Process definition
├── requirements.txt      # Python dependencies (pinned)
├── Dockerfile            # Container definition (optional)
├── .env.example          # Environment template
└── .gitignore            # Excludes secrets
```

---

## 9.3 Configuration Files

**railway.json:**

json
```json
{
  "$schema": "https://railway.app/railway.schema.json",
  "build": {
    "builder": "NIXPACKS",
    "watchPatterns": [
      "requirements.txt",
      "main.py",
      "services/**",
      "railway.json"
    ]
  },
  "deploy": {
    "startCommand": "uvicorn main:app --host 0.0.0.0 --port $PORT --timeout-keep-alive 75",
    "healthcheckPath": "/health",
    "healthcheckTimeout": 300
  }
}
```

**Procfile (alternative):**
```
web: uvicorn main:app --host 0.0.0.0 --port $PORT --timeout-keep-alive 75
```

**requirements.txt:**
```
fastapi==0.115.0
uvicorn[standard]==0.30.0
google-generativeai==0.8.0
pydantic==2.9.0
python-dotenv==1.0.1
structlog==24.4.0
tenacity==9.0.0
jsonrepair==0.1.0
```

**Dockerfile (optional, for more control):**

```dockerfile
FROM python:3.11-slim

WORKDIR /app

# Install dependencies
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copy application
COPY . .

# Expose Railway's dynamic port
EXPOSE $PORT

# Start with graceful shutdown support
```

```
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "$PORT", "--timeout-keep-alive", "75"]
```

---

### 9.4 Environment Variables

**Required Variables:**

| Variable | Required | Description | Example |
|---|---|---|---|
| `GEMINI_API_KEY` | ✅ | Google AI API key | `AIza...` |
| `RAILWAY_DEPLOYMENT_DRAINING_SECONDS` | ✅ | Grace period for in-flight requests | `120` |
| `PORT` | Auto | Railway-injected port | (auto) |

**Optional Variables:**

| Variable | Default | Description |
|---|---|---|
| `VISUAL_WEIGHT` | `60` | Visual component weight |
| `PERSONALITY_WEIGHT` | `30` | Personality component weight |
| `HLA_WEIGHT` | `10` | Genetic component weight |
| `ENVIRONMENT` | `production` | Deployment environment |
| `LOG_LEVEL` | `INFO` | Logging verbosity |
| `GEMINI_MODEL_PRIMARY` | `gemini-3-flash-preview` | Primary model |
| `GEMINI_MODEL_FALLBACK` | `gemini-3-pro-preview` | Fallback model |
| `REQUEST_TIMEOUT` | `70` | FastAPI request timeout (seconds) |

**Setting Variables in Railway:**

1. Navigate to Service → Variables tab
2. Click "New Variable" or use RAW Editor
3. For sensitive values (API keys), enable "Sealed" to hide after save
4. Click "Deploy" to apply changes

⚠️ **CRITICAL:** Always set `RAILWAY_DEPLOYMENT_DRAINING_SECONDS=120` for Harmonia. Without this, 30-60 second Gemini calls will be terminated during redeployment.

---

## 9.5 Health Check Configuration

Railway performs health checks only at deployment start to verify the service is ready before routing traffic. The health check is **not** used for continuous monitoring.

**Health Check Strategy for AI Services:**

| Check Type | Endpoint | Purpose | Gemini Call? |
|---|---|---|---|
| Liveness | `/health` | App initialized and responding | ❌ No |
| Deep (manual) | `/health/deep` | Full system verification | ✅ Yes |

**Why NOT check Gemini in liveness:**

- Gemini API can be temporarily unavailable (rate limits, overload)
- Health check would fail, preventing deployment
- False negatives cause deployment failures for transient issues

**Implementation:**

python
```
from fastapi import FastAPI, Response
from datetime import datetime

app = FastAPI()

# Startup state
startup_time = None

@app.on_event("startup")
async def startup():
    global startup_time
    startup_time = datetime.utcnow()
```

```python
@app.get("/health")
async def health_check():
    """
    Liveness check - Railway uses this.
    Returns 200 if app is initialized. Does NOT check Gemini.
    """
    return {
        "status": "healthy",
        "startup_time": startup_time.isoformat() if startup_time else None,
        "checks": {
            "app_initialized": True,
            "gemini_configured": bool(os.getenv("GEMINI_API_KEY"))
        }
    }

@app.get("/health/deep")
async def deep_health_check():
    """
    Deep check - for manual monitoring only.
    Actually calls Gemini API. Takes 2-3 seconds.
    DO NOT use as Railway health check endpoint.
    """
    try:
        # Quick Gemini ping
        response = await gemini_service.ping()
        gemini_status = "healthy"
    except Exception as e:
        gemini_status = f"unhealthy: {str(e)}"

    return {
        "status": "healthy" if gemini_status == "healthy" else "degraded",
        "checks": {
            "app": "healthy",
            "gemini_api": gemini_status
        }
    }
```

**Health Check Timeout:**

Default timeout is 300 seconds (5 minutes). For Harmonia, the default is sufficient since the `/health` endpoint returns immediately.

json

```
{
  "deploy": {
    "healthcheckPath": "/health",
    "healthcheckTimeout": 300
  }
}
```

---

### 9.6 Timeout Configuration

Gemini API calls can take 10-60 seconds. Timeout configuration must account for this at every layer:

| Layer | Timeout | Rationale |
|---|---|---|
| Gemini API call | 60s | Maximum expected response time |
| FastAPI request | 70s | Gemini (60s) + buffer (10s) |
| HTTP client | 75s | FastAPI (70s) + buffer (5s) |
| Keep-alive | 75s | Match client timeout |
| Uvicorn graceful shutdown | 120s | Allow in-flight requests to complete |

**FastAPI Timeout Middleware:**

python
```
import asyncio
from fastapi import FastAPI, Request, HTTPException
from starlette.middleware.base import BaseHTTPMiddleware

class TimeoutMiddleware(BaseHTTPMiddleware):
    def __init__(self, app, timeout: int = 70):
        super().__init__(app)
        self.timeout = timeout

    async def dispatch(self, request: Request, call_next):
        try:
            return await asyncio.wait_for(
                call_next(request),
                timeout=self.timeout
            )
        except asyncio.TimeoutError:
```

```python
            raise HTTPException(
                status_code=504,
                detail=f"Request timeout after {self.timeout}s"
            )

app = FastAPI()
app.add_middleware(TimeoutMiddleware, timeout=70)
```

**Uvicorn Configuration:**

bash
```bash
uvicorn main:app --host 0.0.0.0 --port $PORT --timeout-keep-alive 75
```

---

### 9.7 Graceful Shutdown

** ⚠️ CRITICAL FOR HARMONIA**

Without graceful shutdown, in-flight Gemini API calls (30-60 seconds) will be terminated during redeployment, causing request failures and poor user experience.

**How Railway Handles Shutdown:**

1. New deployment starts
2. Railway sends `SIGTERM` to old deployment
3. Old deployment has `RAILWAY_DEPLOYMENT_DRAINING_SECONDS` to finish requests
4. After grace period, Railway sends `SIGKILL`
5. Traffic switches to new deployment

**Required Environment Variable:**
```
RAILWAY_DEPLOYMENT_DRAINING_SECONDS=120
```

**FastAPI Graceful Shutdown Implementation:**

python
```python
import signal
import asyncio
from fastapi import FastAPI
from contextlib import asynccontextmanager
import structlog
```

```python
logger = structlog.get_logger()

# Track active requests
active_requests = set()
shutdown_event = asyncio.Event()

@asynccontextmanager
async def lifespan(app: FastAPI):
    """Manage application lifecycle with graceful shutdown."""

    # Startup
    logger.info("application_starting")
    yield

    # Shutdown
    logger.info(
        "shutdown_initiated",
        active_requests=len(active_requests)
    )

    # Wait for active requests to complete (up to 120s)
    if active_requests:
        logger.info(
            "waiting_for_requests",
            count=len(active_requests)
        )

        # Wait up to 110 seconds (leaving 10s buffer before SIGKILL)
        try:
            await asyncio.wait_for(
                wait_for_requests_to_complete(),
                timeout=110
            )
        except asyncio.TimeoutError:
            logger.warning(
                "shutdown_timeout",
                remaining_requests=len(active_requests)
            )

    logger.info("shutdown_complete")

async def wait_for_requests_to_complete():
    """Wait until all active requests finish."""
    while active_requests:
        await asyncio.sleep(0.5)

app = FastAPI(lifespan=lifespan)

@app.middleware("http")
async def track_requests(request, call_next):
    """Track active requests for graceful shutdown."""
    request_id = str(uuid.uuid4())
    active_requests.add(request_id)

    try:
        response = await call_next(request)
        return response
    finally:
        active_requests.discard(request_id)
```

---

## 9.8 Structured Logging

Railway's Log Explorer fully indexes JSON logs, enabling powerful queries on custom fields.

**Log Format:**

python
```
import structlog
import logging
import sys

def configure_logging():
    """Configure structured JSON logging for Railway."""

    structlog.configure(
        processors=[
            structlog.contextvars.merge_contextvars,
            structlog.processors.add_log_level,
            structlog.processors.TimeStamper(fmt="iso"),
            structlog.processors.JSONRenderer()
        ],
        wrapper_class=structlog.make_filtering_bound_logger(logging.INFO),
        context_class=dict,
        logger_factory=structlog.PrintLoggerFactory(),
        cache_logger_on_first_use=True,
    )

configure_logging()
logger = structlog.get_logger()
```

**Logging Gemini Calls:**

python
```
async def parse_response(self, question: str, answer: str, user_id: str):
    """Parse with structured logging for Railway observability."""

    request_id = str(uuid.uuid4())
    start_time = time.time()

    logger.info(
        "gemini_request_start",
        **{
            "@request_id": request_id,
            "@user_id": user_id,
            "@gemini_model": self.primary_model.value,
            "@question_length": len(question),
            "@answer_length": len(answer)
        }
    )

    try:
        result = await self._call_gemini(question, answer)
        latency_ms = int((time.time() - start_time) * 1000)

        logger.info(
            "gemini_request_success",
            **{
                "@request_id": request_id,
                "@user_id": user_id,
                "@gemini_latency_ms": latency_ms,
                "@gemini_model": self.primary_model.value,
                "@tokens_input": result.get("input_tokens", 0),
                "@tokens_output": result.get("output_tokens", 0)
            }
        )

        return result

    except Exception as e:
```

```
    latency_ms = int((time.time() - start_time) * 1000)

    logger.error(
        "gemini_request_failed",
        **{
            "@request_id": request_id,
            "@user_id": user_id,
            "@gemini_latency_ms": latency_ms,
            "@error_type": type(e).__name__,
            "@error_message": str(e)
        }
    )
    raise
```

**Railway Log Explorer Queries:**

| Query | Purpose |
|---|---|
| `@level:error` | All errors |
| `@gemini_latency_ms:>30000` | Gemini calls over 30 seconds |
| `@user_id:user_123` | All logs for specific user |
| `@gemini_model:gemini-3-flash-preview` | Logs for specific model |
| `gemini_request_failed` | All failed Gemini requests |

---

## 9.9 Monitoring and Alerting

**Built-in Metrics (Railway Dashboard):**

| Metric | Description | Available |
|---|---|---|
| CPU Usage | Per-service and per-replica | ✅ All tiers |
| Memory Usage | RAM consumption over time | ✅ All tiers |
| Request Count | Broken down by status code | ✅ All tiers |
| Network Egress | Outbound data transfer | ✅ All tiers |

**Alerting (Pro Tier Only):**

Configure monitors from any widget in the Observability Dashboard:

| Metric | Warning | Critical | Action |
|---|---|---|---|
| CPU | >70% sustained | >90% sustained | Scale replicas |
| Memory | >80% | >95% | Increase allocation |
| Error rate (5xx) | >2% | >5% | Investigate logs |
| P95 latency | >45s | >60s | Check Gemini status |

**Webhook Integration:**

json
~~~
{
  "webhook_url": "https://hooks.slack.com/services/xxx",
  "events": ["deployment_success", "deployment_failure", "monitor_triggered"]
}
```


---

### 9.10 Cost Estimation

**Harmonia Workload Profile:**

| Parameter | Value |
|-----------|-------|
| Users per day | 1,000 |
| Gemini calls per user | 6 |
| Average call duration | 30 seconds |
| Total API calls/day | 6,000 |
| Total compute time/day | 50 hours (I/O wait, not CPU) |

**Monthly Cost Breakdown (Pro Tier):**

| Component | Cost | Notes |
|-----------|------|-------|
| Pro subscription | $20 | Base plan |
| RAM (1 GB) | $10 | `$10/GB/month` |
~~~

| CPU (1 vCPU) | $20 | `$20/vCPU/month` |
| Network egress | <$1 | Gemini responses are small |
| **Railway Total** | **~$50-55** | |

**Gemini API Cost (Separate):**

| Model | Input (per 1M tokens) | Output (per 1M tokens) |
|-------|----------------------|-----------------------|
| Flash | $0.50 | $3.00 |
| Pro | $2.00 | $12.00 |

At 6,000 calls/day × 5K input tokens × 2K output tokens:
- Flash: ~$1-2/month
- Pro: ~$5-10/month

**Total Monthly Cost:** ~$55-65 (Railway + Gemini API)

**Cost Per User:** ~$0.05-0.07/month

---

### 9.11 Private Networking (Multi-Service Architecture)

If Harmonia scales to separate API and worker services:

**Architecture:**
```
┌─────────────────────────────────────────────────────────────┐
│              Railway Project                  │
│                                          │
│  ┌──────────────────┐      ┌────────────────────────┐    │
│  │ API Service │───────→│  Worker Service  │   │
│  │ (FastAPI)   │ IPv6  │  (Background)    │   │
│  │ Port 8080   │  5ms  │   Port 8081      │   │
│  └──────────────────┘      └────────────────────────┘    │
│      │             │         │
│      ▼             ▼           │
│  ┌──────────────────┐      ┌────────────────────────┐    │
│  │  Postgres   │      │  Redis       │   │
│  │ (Database)  │      │ (Job Queue)    │   │
│  └──────────────────┘      └────────────────────────┘    │
│                                          │
└─────────────────────────────────────────────────────────────┘
```

**Service Discovery:**

Services communicate via internal DNS using the pattern:

```
<service-name>.railway.internal
```

**Example:**

python

```
# In API service, call worker service
WORKER_URL = "http://worker.railway.internal:8081"

async def queue_analysis(user_id: str, questions: list):
    async with httpx.AsyncClient() as client:
        response = await client.post(
            f"{WORKER_URL}/analyze",
            json={"user_id": user_id, "questions": questions}
        )
    return response.json()
```

**IPv6 Binding:**

For private networking, services must bind to IPv6:

bash

```
uvicorn main:app --host "::" --port $PORT
```

Or in Python:

python

```
uvicorn.run(app, host="::", port=int(os.getenv("PORT", 8080)))
```

---

## 9.12 Deployment Checklist

**Pre-Deployment:**

- `GEMINI_API_KEY` set in Railway Variables (sealed)
- `RAILWAY_DEPLOYMENT_DRAINING_SECONDS=120` configured
- Health check endpoint (`/health`) implemented and tested
- Graceful shutdown handler implemented

- Structured logging configured (JSON format)
- `.gitignore` excludes `.env`, secrets, API keys
- `requirements.txt` has pinned versions
- Local testing passes with production environment variables

**Post-Deployment:**

- Health check passes (deployment succeeds)
- `/health` endpoint returns 200
- Test Gemini API call completes successfully
- Logs appear in Railway Log Explorer
- Custom fields (`@gemini_latency_ms`) are queryable
- Monitoring dashboard configured (Pro tier)
- Webhook notifications working (Slack/Discord)

**Production Validation:**

- Cold start time acceptable (<30s for Starter, none for Pro)
- 60-second Gemini calls complete without timeout
- Redeployment doesn't kill in-flight requests
- Error rate <1% over 24 hours
- P95 latency within acceptable range

---

## 9.13 Summary

**Critical Configuration:**

| Setting | Value | Why |
|---|---|---|
| `RAILWAY_DEPLOYMENT_DRAINING_SECONDS` | `120` | Prevents request loss during redeploy |
| Health check path | `/health` | Liveness only, no Gemini call |
| Uvicorn keep-alive | `75s` | Matches client timeout |
| FastAPI request timeout | `70s` | Gemini (60s) + buffer |

**Tier Recommendation:**

| Phase | Tier | Monthly Cost | Reason |
|---|---|---|---|

| | | | |
|---|---|---|---|
| Beta (20-100 users) | Starter | ~$5-10 | Acceptable cold starts |
| Production (100+ users) | Pro | ~$50-55 | Always-on, monitoring, SLA |

**What This Section Achieves:**

1. Complete Railway configuration for long-running AI workloads
2. Graceful shutdown preventing request loss during redeployment
3. Structured logging enabling Gemini latency tracking
4. Cost estimation for budget planning
5. Health check patterns appropriate for external API dependencies
6. Private networking setup for multi-service scaling

**Honest Limitations:**

- Railway Starter tier has cold starts (10-30 seconds after inactivity)
- No native circuit breaker — must implement in application code
- Log retention limited to 30 days
- Monitoring alerts require Pro tier

## Section 10: Testing

This section documents the complete testing strategy for Harmonia's personality analysis pipeline, including unit tests, integration tests, fixtures, and validation criteria with specific input/output examples.

### 10.1 Test Architecture Overview

Test Directory Structure:

```
harmonia/
├── services/
│   ├── gemini_service.py
│   ├── profile_service.py
│   └── similarity_service.py
├── tests/
│   ├── __init__.py
│   ├── conftest.py              # Shared fixtures
│   ├── unit/
│   │   ├── __init__.py
│   │   ├── test_gemini_service.py
│   │   ├── test_profile_service.py
│   │   └── test_similarity_service.py
│   ├── integration/
│   │   ├── __init__.py
│   │   ├── test_pipeline.py
│   │   └── test_api_endpoints.py
│   └── fixtures/
│       ├── sample_responses.json
│       ├── sample_profiles.json
│       └── mock_gemini_responses.json
├── pytest.ini
└── requirements-test.txt
```

Testing Dependencies (requirements-test.txt):

```
pytest==8.3.0
pytest-asyncio==0.24.0
pytest-cov==5.0.0
pytest-mock==3.14.0
httpx==0.27.0
respx==0.21.0
freezegun==1.4.0
factory-boy==3.3.0
```

pytest.ini Configuration:

```ini
ini[pytest]
asyncio_mode = auto
testpaths = tests
python_files = test_*.py
python_functions = test_*
addopts = -v --tb=short --cov=services --cov-report=term-missing
markers =
    unit: Unit tests (fast, no external dependencies)
    integration: Integration tests (may require services)
    slow: Slow tests (skip with -m "not slow")
```

### 10.2 Shared Fixtures (conftest.py)

```python
python# tests/conftest.py

import pytest
import json
from unittest.mock import AsyncMock, MagicMock
from datetime import datetime

# ═══════════════════════════════════════════════════════════════════════════════
#                              SAMPLE DATA FIXTURES
# ═══════════════════════════════════════════════════════════════════════════════

@pytest.fixture
```

```
def sample_question():
    """Sample Felix questionnaire question."""
    return "You and friends just finished dinner at a restaurant. Everyone
contributed differently to the total. What's your approach?"


@pytest.fixture
def sample_answer_cooperative():
    """Sample answer indicating cooperative personality (low wrath, low
greed)."""
    return """
    I usually suggest we split evenly because it keeps things simple and nobody
    feels awkward. But if someone clearly ordered way less, I'd offer to cover
    more of my share. Money isn't worth making friends uncomfortable over. I'd
    rather pay a bit extra than have anyone feel like they're being taken
    advantage of or create tension in the group.
    """


@pytest.fixture
def sample_answer_assertive():
    """Sample answer indicating assertive personality (high wrath, high
pride)."""
    return """
    I always track what I ordered and pay exactly that. Why should I subsidize
    someone else's expensive steak when I had a salad? I'm not afraid to speak
    up about it either - it's about fairness. If people get uncomfortable,
    that's their problem. I've worked hard for my money and I'm not going to
    let social pressure make me overpay.
    """


@pytest.fixture
def sample_answer_minimal():
    """Minimal answer that should trigger low confidence scores."""
    return "Split it evenly I guess."


@pytest.fixture
def sample_answer_contradictory():
    """Answer with contradictory signals (should trigger discrepancy flags)."""
    return """
    I'm always super chill about money, never get angry about anything. But
    honestly it INFURIATES me when people don't pay their fair share! I
absolutely
    HATE freeloaders. Still, I'm the most easygoing person you'll ever meet and
    I never let things bother me.
    """


# ═══════════════════════════════════════════════════════════════════════════════
#                              PARSED RESPONSE FIXTURES
# ═══════════════════════════════════════════════════════════════════════════════

@pytest.fixture
def parsed_response_cooperative():
    """Expected Gemini output for cooperative answer."""
    return {
        "sins": {
            "greed": {"score": -2.0, "confidence": 0.85, "evidence": "Money
isn't worth making friends uncomfortable"},
            "pride": {"score": 0.5, "confidence": 0.60, "evidence": ""},
            "lust": {"score": 0.0, "confidence": 0.50, "evidence": ""},
            "wrath": {"score": -3.0, "confidence": 0.90, "evidence": "Rather pay
```

```
extra than create tension"},
            "gluttony": {"score": 0.0, "confidence": 0.50, "evidence": ""},
            "envy": {"score": -1.0, "confidence": 0.65, "evidence": "Not
comparative"},
            "sloth": {"score": 1.0, "confidence": 0.70, "evidence": "Prefers
simple solution"}
        },
        "liwc_signals": {
            "first_person_singular": 0.08,
            "first_person_plural": 0.02,
            "negative_emotion": 0.01,
            "positive_emotion": 0.02,
            "certainty": 0.01,
            "hedging": 0.02
        },
        "discrepancies": [],
        "word_count": 75
    }


@pytest.fixture
def parsed_response_assertive():
    """Expected Gemini output for assertive answer."""
    return {
        "sins": {
            "greed": {"score": 2.5, "confidence": 0.80, "evidence": "Track what
I ordered and pay exactly that"},
            "pride": {"score": 3.0, "confidence": 0.85, "evidence": "I've worked
hard for my money"},
            "lust": {"score": 0.0, "confidence": 0.50, "evidence": ""},
            "wrath": {"score": 2.5, "confidence": 0.88, "evidence": "Not afraid
to speak up"},
            "gluttony": {"score": 0.0, "confidence": 0.50, "evidence": ""},
            "envy": {"score": 1.0, "confidence": 0.60, "evidence": "Subsidize
someone else's expensive steak"},
            "sloth": {"score": -1.0, "confidence": 0.65, "evidence": "Always
track what I ordered"}
        },
        "liwc_signals": {
            "first_person_singular": 0.12,
            "first_person_plural": 0.00,
            "negative_emotion": 0.03,
            "positive_emotion": 0.00,
            "certainty": 0.04,
            "hedging": 0.00
        },
        "discrepancies": [],
        "word_count": 82
    }


# ═══════════════════════════════════════════════════════════════════════════
#                               PROFILE FIXTURES
# ═══════════════════════════════════════════════════════════════════════════

@pytest.fixture
def profile_user_a():
    """Complete profile for User A (cooperative type)."""
    return {
        "user_id": "user_a_123",
        "version": 1,
        "created_at": "2026-01-29T14:30:00Z",
        "sins": {
            "greed": {"score": -1.8, "confidence": 0.82, "variance": 1.2,
"high_variance_flag": False},
            "pride": {"score": 0.3, "confidence": 0.58, "variance": 2.1,
"high_variance_flag": False},
            "lust": {"score": 1.5, "confidence": 0.75, "variance": 1.8,
"high_variance_flag": False},
            "wrath": {"score": -2.5, "confidence": 0.88, "variance": 0.9,
"high_variance_flag": False},
            "gluttony": {"score": 0.8, "confidence": 0.62, "variance": 2.5,
"high_variance_flag": False},
            "envy": {"score": -0.5, "confidence": 0.68, "variance": 1.5,
"high_variance_flag": False},
            "sloth": {"score": 0.2, "confidence": 0.70, "variance": 1.2,
"high_variance_flag": False}
        },
        "quality": {
            "score": 78.5,
            "tier": "moderate"
        }
    }


@pytest.fixture
def profile_user_b_similar():
    """Profile for User B with high similarity to User A."""
    return {
        "user_id": "user_b_456",
        "version": 1,
        "created_at": "2026-01-29T15:00:00Z",
        "sins": {
            "greed": {"score": -2.2, "confidence": 0.80, "variance": 1.5,
"high_variance_flag": False},
            "pride": {"score": 0.8, "confidence": 0.65, "variance": 1.8,
"high_variance_flag": False},
            "lust": {"score": 2.0, "confidence": 0.78, "variance": 1.2,
"high_variance_flag": False},
            "wrath": {"score": -3.0, "confidence": 0.92, "variance": 0.7,
"high_variance_flag": False},
            "gluttony": {"score": 1.2, "confidence": 0.60, "variance": 2.8,
"high_variance_flag": False},
            "envy": {"score": -0.8, "confidence": 0.72, "variance": 1.3,
"high_variance_flag": False},
            "sloth": {"score": 0.5, "confidence": 0.68, "variance": 1.5,
"high_variance_flag": False}
        },
        "quality": {
            "score": 81.2,
            "tier": "high"
        }
    }


@pytest.fixture
def profile_user_c_opposite():
    """Profile for User C with low similarity to User A (opposite traits)."""
    return {
        "user_id": "user_c_789",
        "version": 1,
        "created_at": "2026-01-29T15:30:00Z",
        "sins": {
            "greed": {"score": 2.5, "confidence": 0.85, "variance": 1.0,
"high_variance_flag": False},
            "pride": {"score": 3.2, "confidence": 0.82, "variance": 1.5,
"high_variance_flag": False},
            "lust": {"score": -1.5, "confidence": 0.70, "variance": 2.0,
"high_variance_flag": False},
            "wrath": {"score": 3.0, "confidence": 0.90, "variance": 0.8,
"high_variance_flag": False},
            "gluttony": {"score": -0.5, "confidence": 0.55, "variance": 3.2,
"high_variance_flag": True},
            "envy": {"score": 2.0, "confidence": 0.78, "variance": 1.2,
"high_variance_flag": False},
            "sloth": {"score": -2.0, "confidence": 0.75, "variance": 1.8,
"high_variance_flag": False}
        },
        "quality": {
            "score": 75.0,
            "tier": "moderate"
        }
    }


# ═══════════════════════════════════════════════════════════════════════════
#                              MOCK SERVICE FIXTURES
# ═══════════════════════════════════════════════════════════════════════════

@pytest.fixture
def mock_gemini_service(parsed_response_cooperative):
    """Mock GeminiService that returns predictable responses."""
    service = MagicMock()
    service.parse_response = AsyncMock(return_value=parsed_response_cooperative)
    return service


@pytest.fixture
def mock_gemini_service_factory():
    """Factory to create mock GeminiService with custom responses."""
    def _create(response_data):
        service = MagicMock()
        service.parse_response = AsyncMock(return_value=response_data)
        return service
    return _create


# ═══════════════════════════════════════════════════════════════════════════
#                              SERVICE INSTANCE FIXTURES
# ═══════════════════════════════════════════════════════════════════════════

@pytest.fixture
def profile_service():
    """Real ProfileService instance for unit testing."""
    from services.profile_service import ProfileService
    return ProfileService()


@pytest.fixture
def similarity_service():
    """Real SimilarityService instance for unit testing."""
    from services.similarity_service import SimilarityService
    return SimilarityService()
```

10.3 Unit Tests: GeminiService

```
python# tests/unit/test_gemini_service.py

import pytest
import json
from unittest.mock import AsyncMock, MagicMock, patch
```

```
class TestGeminiServiceValidation:
    """Tests for Gemini response validation."""

    def test_validate_score_range_valid(self):
        """Scores within -5 to +5 should pass validation."""
        from services.gemini_service import GeminiService

        valid_scores = [-5, -3.5, 0, 2.5, 5]
        for score in valid_scores:
            response = {
                "sins": {
                    sin: {"score": score, "confidence": 0.8, "evidence": ""}
                    for sin in ["greed", "pride", "lust", "wrath", "gluttony",
"envy", "sloth"]
                }
            }
            # Should not raise
            GeminiService._validate_parsed_response(None, response)

    def test_validate_score_range_invalid(self):
        """Scores outside -5 to +5 should raise ValueError."""
        from services.gemini_service import GeminiService

        invalid_scores = [-6, -5.1, 5.1, 6, 10]
        for score in invalid_scores:
            response = {
                "sins": {
                    "greed": {"score": score, "confidence": 0.8, "evidence":
""},
                    "pride": {"score": 0, "confidence": 0.8, "evidence": ""},
                    "lust": {"score": 0, "confidence": 0.8, "evidence": ""},
                    "wrath": {"score": 0, "confidence": 0.8, "evidence": ""},
                    "gluttony": {"score": 0, "confidence": 0.8, "evidence": ""},
                    "envy": {"score": 0, "confidence": 0.8, "evidence": ""},
                    "sloth": {"score": 0, "confidence": 0.8, "evidence": ""}
                }
            }
            with pytest.raises(ValueError, match="Invalid score"):
                GeminiService._validate_parsed_response(None, response)

    def test_validate_confidence_range_valid(self):
        """Confidence within 0.0 to 1.0 should pass validation."""
        from services.gemini_service import GeminiService

        valid_confidences = [0.0, 0.5, 0.85, 1.0]
        for conf in valid_confidences:
            response = {
                "sins": {
                    sin: {"score": 0, "confidence": conf, "evidence": ""}
                    for sin in ["greed", "pride", "lust", "wrath", "gluttony",
"envy", "sloth"]
                }
            }
            # Should not raise
            GeminiService._validate_parsed_response(None, response)

    def test_validate_confidence_range_invalid(self):
        """Confidence outside 0.0 to 1.0 should raise ValueError."""
        from services.gemini_service import GeminiService

        invalid_confidences = [-0.1, 1.1, 2.0]
        for conf in invalid_confidences:
            response = {
                "sins": {
```

~~~
                    "greed": {"score": 0, "confidence": conf, "evidence": ""},
                    "pride": {"score": 0, "confidence": 0.8, "evidence": ""},
                    "lust": {"score": 0, "confidence": 0.8, "evidence": ""},
                    "wrath": {"score": 0, "confidence": 0.8, "evidence": ""},
                    "gluttony": {"score": 0, "confidence": 0.8, "evidence": ""},
                    "envy": {"score": 0, "confidence": 0.8, "evidence": ""},
                    "sloth": {"score": 0, "confidence": 0.8, "evidence": ""}
                }
            }
            with pytest.raises(ValueError, match="Invalid confidence"):
                GeminiService._validate_parsed_response(None, response)

    def test_validate_missing_sins(self):
        """Missing sins should raise ValueError."""
        from services.gemini_service import GeminiService

        # Missing 'wrath'
        response = {
            "sins": {
                "greed": {"score": 0, "confidence": 0.8, "evidence": ""},
                "pride": {"score": 0, "confidence": 0.8, "evidence": ""},
                "lust": {"score": 0, "confidence": 0.8, "evidence": ""},
                # wrath missing
                "gluttony": {"score": 0, "confidence": 0.8, "evidence": ""},
                "envy": {"score": 0, "confidence": 0.8, "evidence": ""},
                "sloth": {"score": 0, "confidence": 0.8, "evidence": ""}
            }
        }
        with pytest.raises(ValueError, match="Missing required sins"):
            GeminiService._validate_parsed_response(None, response)


class TestGeminiServiceJSONExtraction:
    """Tests for JSON extraction from Gemini responses."""

    def test_extract_clean_json(self):
        """Clean JSON should parse directly."""
        from services.gemini_service import GeminiService

        text = '{"sins": {"greed": {"score": 0, "confidence": 0.8}}}'
        result = GeminiService._extract_json_from_response(None, text)
        assert result["sins"]["greed"]["score"] == 0

    def test_extract_json_from_markdown(self):
        """JSON wrapped in markdown code blocks should be extracted."""
        from services.gemini_service import GeminiService

        text = '''```json
{"sins": {"greed": {"score": -2, "confidence": 0.85}}}
```'''
        result = GeminiService._extract_json_from_response(None, text)
        assert result["sins"]["greed"]["score"] == -2

    def test_extract_json_with_preamble(self):
        """JSON with preamble text should be extracted."""
        from services.gemini_service import GeminiService

        text = '''Here's the analysis:
```json
{"sins": {"greed": {"score": 1, "confidence": 0.75}}}
~~~
This shows moderate greed.'''
        result = GeminiService._extract_json_from_response(None, text)
        assert result["sins"]["greed"]["score"] == 1
```
```

```
    def test_extract_invalid_json_raises(self):
        """Invalid JSON should raise ValueError."""
        from services.gemini_service import GeminiService

        text = '{"sins": {invalid json here}'
        with pytest.raises(ValueError, match="Invalid JSON"):
            GeminiService._extract_json_from_response(None, text)


class TestGeminiServiceLIWC:
    """Tests for LIWC signal extraction."""

    def test_liwc_first_person_singular(self):
        """First person singular pronouns should be counted correctly."""
        from services.gemini_service import GeminiService

        text = "I think I should pay my share because I ordered it"
        # 4 first-person singular pronouns (I, I, my, I) out of 10 words
        result = GeminiService._extract_liwc_signals(None, text)
        assert 0.35 <= result["first_person_singular"] <= 0.45

    def test_liwc_negative_emotion(self):
        """Negative emotion words should be counted."""
        from services.gemini_service import GeminiService

        text = "I hate when people are angry and frustrated with me"
        result = GeminiService._extract_liwc_signals(None, text)
        assert result["negative_emotion"] > 0.1  # hate, angry, frustrated

    def test_liwc_empty_text(self):
        """Empty text should return zeros without crashing."""
        from services.gemini_service import GeminiService

        result = GeminiService._extract_liwc_signals(None, "")
        assert result["first_person_singular"] == 0
        assert result["negative_emotion"] == 0


class TestGeminiServiceDiscrepancies:
    """Tests for discrepancy detection."""

    def test_detect_wrath_discrepancy(self):
        """Wrath discrepancy should be flagged when score contradicts text."""
        from services.gemini_service import GeminiService

        text = "I'm never angry or frustrated. I hate when people annoyed me
though, it makes me furious."
        sins = {
            "wrath": {"score": -3, "confidence": 0.85},  # Claims low wrath
            "greed": {"score": 0, "confidence": 0.5},
            "pride": {"score": 0, "confidence": 0.5},
            "lust": {"score": 0, "confidence": 0.5},
            "gluttony": {"score": 0, "confidence": 0.5},
            "envy": {"score": 0, "confidence": 0.5},
            "sloth": {"score": 0, "confidence": 0.5}
        }

        result = GeminiService._detect_discrepancies(None, text, sins)
        assert "wrath_discrepancy" in result

    def test_detect_pride_discrepancy(self):
        """Pride discrepancy should be flagged when humble claim contradicts
self-promotion."""
```

````python
        from services.gemini_service import GeminiService

        text = "I'm very humble. But honestly I'm the best at this, my work is amazing and incredible."
        sins = {
            "pride": {"score": -3, "confidence": 0.80},  # Claims humble
            "greed": {"score": 0, "confidence": 0.5},
            "wrath": {"score": 0, "confidence": 0.5},
            "lust": {"score": 0, "confidence": 0.5},
            "gluttony": {"score": 0, "confidence": 0.5},
            "envy": {"score": 0, "confidence": 0.5},
            "sloth": {"score": 0, "confidence": 0.5}
        }

        result = GeminiService._detect_discrepancies(None, text, sins)
        assert "pride_discrepancy" in result

    def test_no_discrepancy_when_consistent(self):
        """No discrepancy should be flagged when text matches score."""
        from services.gemini_service import GeminiService

        text = "I always stay calm and prefer to avoid conflict. Harmony is important to me."
        sins = {
            "wrath": {"score": -3, "confidence": 0.90},
            "greed": {"score": 0, "confidence": 0.5},
            "pride": {"score": 0, "confidence": 0.5},
            "lust": {"score": 0, "confidence": 0.5},
            "gluttony": {"score": 0, "confidence": 0.5},
            "envy": {"score": 0, "confidence": 0.5},
            "sloth": {"score": 0, "confidence": 0.5}
        }

        result = GeminiService._detect_discrepancies(None, text, sins)
        assert len(result) == 0
````

---

### 10.4 Unit Tests: ProfileService

````python
# tests/unit/test_profile_service.py

import pytest
import statistics


class TestProfileServiceAggregation:
    """Tests for trait aggregation logic."""

    def test_aggregate_single_trait_simple_mean(self, profile_service):
        """When confidence doesn't vary, use simple mean."""
        scores = [
            {"score": 2.0, "confidence": 0.80, "evidence": ""},
            {"score": 3.0, "confidence": 0.80, "evidence": ""},
            {"score": 2.5, "confidence": 0.80, "evidence": ""},
            {"score": 2.0, "confidence": 0.80, "evidence": ""},
            {"score": 3.0, "confidence": 0.80, "evidence": ""},
            {"score": 2.5, "confidence": 0.80, "evidence": ""}
        ]

        result = profile_service._aggregate_single_trait(scores)

        assert result["method"] == "mean"
````

```python
        assert 2.4 <= result["score"] <= 2.6  # Mean of 2.5
        assert result["item_count"] == 6
        assert result["high_variance_flag"] == False

    def test_aggregate_single_trait_cwmv(self, profile_service):
        """When confidence varies, use confidence-weighted mean."""
        scores = [
            {"score": -3.0, "confidence": 0.95, "evidence": "Strong signal"},
            {"score": 1.0, "confidence": 0.30, "evidence": "Weak signal"},
            {"score": -2.5, "confidence": 0.90, "evidence": "Strong signal"},
            {"score": 0.5, "confidence": 0.40, "evidence": "Weak signal"},
            {"score": -2.0, "confidence": 0.85, "evidence": "Good signal"},
            {"score": -2.5, "confidence": 0.88, "evidence": "Good signal"}
        ]

        result = profile_service._aggregate_single_trait(scores)

        assert result["method"] == "cwmv"
        # High-confidence negative scores should dominate
        assert result["score"] < -1.5
        assert result["item_count"] == 6

    def test_aggregate_high_variance_flag(self, profile_service):
        """High variance (SD > 3.0) should trigger flag."""
        scores = [
            {"score": -5.0, "confidence": 0.80, "evidence": ""},
            {"score": 5.0, "confidence": 0.80, "evidence": ""},
            {"score": -4.0, "confidence": 0.80, "evidence": ""},
            {"score": 4.0, "confidence": 0.80, "evidence": ""},
            {"score": -3.0, "confidence": 0.80, "evidence": ""},
            {"score": 3.0, "confidence": 0.80, "evidence": ""}
        ]

        result = profile_service._aggregate_single_trait(scores)

        # SD of [-5, 5, -4, 4, -3, 3] ≈ 4.2, which is > 3.0
        assert result["high_variance_flag"] == True
        # Confidence should be penalized
        assert result["confidence"] < 0.80

    def test_aggregate_empty_scores(self, profile_service):
        """Empty scores should return default values."""
        result = profile_service._aggregate_single_trait([])

        assert result["score"] == 0.0
        assert result["confidence"] == 0.0
        assert result["item_count"] == 0
        assert result["method"] == "none"


class TestProfileServiceQuality:
    """Tests for profile quality scoring."""

    def test_quality_high_tier(self, profile_service):
        """High-quality profile should score ≥ 80."""
        sins = {
            "greed": {"score": -2.0, "confidence": 0.85, "variance": 1.5},
            "pride": {"score": 1.0, "confidence": 0.80, "variance": 1.2},
            "lust": {"score": 2.5, "confidence": 0.88, "variance": 1.0},
            "wrath": {"score": -1.5, "confidence": 0.90, "variance": 0.8},
            "gluttony": {"score": 0.5, "confidence": 0.82, "variance": 1.8},
            "envy": {"score": -0.5, "confidence": 0.78, "variance": 1.5},
            "sloth": {"score": 0.0, "confidence": 0.85, "variance": 1.2}
        }
        styles = {"ers": {"flag": False}, "mrs": {"flag": False}, "patterns": 
{"flag": False}}

        result = profile_service._calculate_quality_score(
            sins=sins,
            styles=styles,
            word_count=450,
            time=300
        )

        assert result["score"] >= 80
        assert result["tier"] == "high"

    def test_quality_low_tier(self, profile_service):
        """Low-quality profile should score < 60."""
        sins = {
            "greed": {"score": 0.0, "confidence": 0.40, "variance": 5.0},
            "pride": {"score": 0.0, "confidence": 0.35, "variance": 4.5},
            "lust": {"score": 0.0, "confidence": 0.38, "variance": 5.2},
            "wrath": {"score": 0.0, "confidence": 0.42, "variance": 4.8},
            "gluttony": {"score": 0.0, "confidence": 0.36, "variance": 5.5},
            "envy": {"score": 0.0, "confidence": 0.40, "variance": 4.2},
            "sloth": {"score": 0.0, "confidence": 0.38, "variance": 5.0}
        }
        styles = {"ers": {"flag": True}, "mrs": {"flag": True}, "patterns": 
{"flag": True}}

        result = profile_service._calculate_quality_score(
            sins=sins,
            styles=styles,
            word_count=100,  # Low word count
            time=60  # Very fast (suspicious)
        )

        assert result["score"] < 60
        assert result["tier"] == "low"


class TestProfileServiceResponseStyles:
    """Tests for response style detection."""

    def test_detect_ers_extreme_response_style(self, profile_service):
        """ERS should be flagged when >40% of scores are extreme (±5)."""
        all_scores = [
            [{"score": 5, "confidence": 0.8}, {"score": -5, "confidence": 0.8}] 
* 3,  # Q1: all extreme
            [{"score": 5, "confidence": 0.8}, {"score": -5, "confidence": 0.8}] 
* 3,  # Q2: all extreme
            [{"score": 5, "confidence": 0.8}, {"score": -5, "confidence": 0.8}] 
* 3,  # Q3: all extreme
            [{"score": 0, "confidence": 0.5}] * 7,  # Q4: neutral
            [{"score": 0, "confidence": 0.5}] * 7,  # Q5: neutral
            [{"score": 0, "confidence": 0.5}] * 7   # Q6: neutral
        ]

        # Flatten for ERS calculation
        flat_scores = [s["score"] for q in all_scores for s in q]
        extreme_count = sum(1 for s in flat_scores if abs(s) == 5)
        total = len(flat_scores)

        result = profile_service._detect_ers(all_scores)

        # Should flag ERS since many extreme scores
        assert result["percentage"] > 30
    def test_detect_mrs_midpoint_response_style(self, profile_service):
        """MRS should be flagged when >50% scores are 0 with fast completion."""
        all_scores = [
            [{"score": 0, "confidence": 0.5}] * 7,  # Q1: all neutral
            [{"score": 0, "confidence": 0.5}] * 7,  # Q2: all neutral
            [{"score": 0, "confidence": 0.5}] * 7,  # Q3: all neutral
            [{"score": 0, "confidence": 0.5}] * 7,  # Q4: all neutral
            [{"score": 1, "confidence": 0.6}] * 7,  # Q5: slight variation
            [{"score": -1, "confidence": 0.6}] * 7  # Q6: slight variation
        ]

        result = profile_service._detect_mrs(all_scores, completion_time=90)  #
Fast completion

        # Should flag MRS since >50% neutral + fast completion
        assert result["percentage"] > 50


class TestProfileServiceMinimumRequirements:
    """Tests for minimum viable profile requirements."""

    def test_reject_insufficient_questions(self, profile_service):
        """Profile should be rejected if < 4 valid questions."""
        parsed_responses = [
            {"sins": {"greed": {"score": 0, "confidence": 0.5}}, "word_count":
30},
            {"sins": {"greed": {"score": 0, "confidence": 0.5}}, "word_count":
30},
            {"sins": {"greed": {"score": 0, "confidence": 0.5}}, "word_count":
30}
            # Only 3 questions
        ]

        result = profile_service.aggregate_profile(
            user_id="test_user",
            parsed_responses=parsed_responses
        )

        assert result["quality"]["tier"] == "rejected"
        assert "insufficient_responses" in result["flags"]

    def test_accept_minimum_questions(self, profile_service):
        """Profile should be accepted with exactly 4 valid questions."""
        base_sins = {
            sin: {"score": 0, "confidence": 0.7, "evidence": ""}
            for sin in ["greed", "pride", "lust", "wrath", "gluttony", "envy",
"sloth"]
        }

        parsed_responses = [
            {"sins": base_sins.copy(), "word_count": 50, "discrepancies": []},
            {"sins": base_sins.copy(), "word_count": 55, "discrepancies": []},
            {"sins": base_sins.copy(), "word_count": 60, "discrepancies": []},
            {"sins": base_sins.copy(), "word_count": 45, "discrepancies": []}
        ]

        result = profile_service.aggregate_profile(
            user_id="test_user",
            parsed_responses=parsed_responses
        )

        assert result["quality"]["tier"] != "rejected"
```

---

### 10.5 Unit Tests: SimilarityService
````python
# tests/unit/test_similarity_service.py

import pytest

class TestSimilarityServiceBasic:
    """Basic similarity calculation tests."""

    def test_identical_profiles_maximum_similarity(self, similarity_service):
        """Identical profiles should have maximum similarity."""
        profile_a = {
            "user_id": "a",
            "sins": {
                "greed": {"score": -2.0, "confidence": 0.85},
                "pride": {"score": 1.5, "confidence": 0.80},
                "lust": {"score": 3.0, "confidence": 0.90},
                "wrath": {"score": -3.0, "confidence": 0.88},
                "gluttony": {"score": 0.5, "confidence": 0.70},
                "envy": {"score": -1.5, "confidence": 0.75},
                "sloth": {"score": 2.0, "confidence": 0.82}
            },
            "quality": {"score": 80, "tier": "high"}
        }

        result = similarity_service.calculate_similarity(profile_a, profile_a)

        # Identical profiles should have very high similarity
        assert result["raw_score"] >= 0.90

    def test_opposite_profiles_low_similarity(self, similarity_service,
profile_user_a, profile_user_c_opposite):
        """Opposite profiles should have low similarity (no positive
overlap)."""
        result = similarity_service.calculate_similarity(profile_user_a,
profile_user_c_opposite)

        # Profiles with opposite trait directions should have minimal overlap
        assert result["overlap_count"] <= 2
        assert result["adjusted_score"] < 0.30

    def test_similar_profiles_high_similarity(self, similarity_service,
profile_user_a, profile_user_b_similar):
        """Similar profiles should have high similarity."""
        result = similarity_service.calculate_similarity(profile_user_a,
profile_user_b_similar)

        # Both users have similar directions on most traits
        assert result["overlap_count"] >= 3
        assert result["adjusted_score"] >= 0.40

class TestSimilarityServicePositiveOverlap:
    """Tests for positive overlap calculation logic."""

    def test_both_vice_creates_overlap(self, similarity_service):
        """Two high vice scores on same trait should create overlap."""
        profile_a = {
            "user_id": "a",
            "sins": {
````

```
                "pride": {"score": 3.0, "confidence": 0.85},  # High vice
                "greed": {"score": 0, "confidence": 0.5},
                "lust": {"score": 0, "confidence": 0.5},
                "wrath": {"score": 0, "confidence": 0.5},
                "gluttony": {"score": 0, "confidence": 0.5},
                "envy": {"score": 0, "confidence": 0.5},
                "sloth": {"score": 0, "confidence": 0.5}
            },
            "quality": {"score": 80, "tier": "high"}
        }
        profile_b = {
            "user_id": "b",
            "sins": {
                "pride": {"score": 2.5, "confidence": 0.80},  # High vice
                "greed": {"score": 0, "confidence": 0.5},
                "lust": {"score": 0, "confidence": 0.5},
                "wrath": {"score": 0, "confidence": 0.5},
                "gluttony": {"score": 0, "confidence": 0.5},
                "envy": {"score": 0, "confidence": 0.5},
                "sloth": {"score": 0, "confidence": 0.5}
            },
            "quality": {"score": 80, "tier": "high"}
        }

        result = similarity_service.calculate_similarity(profile_a, profile_b)

        # Should find pride overlap
        overlapping_sins = [b["sin"] for b in result["breakdown"]]
        assert "pride" in overlapping_sins

    def test_both_virtue_creates_overlap(self, similarity_service):
        """Two high virtue scores on same trait should create overlap."""
        profile_a = {
            "user_id": "a",
            "sins": {
                "wrath": {"score": -3.0, "confidence": 0.90},  # High virtue
                "greed": {"score": 0, "confidence": 0.5},
                "pride": {"score": 0, "confidence": 0.5},
                "lust": {"score": 0, "confidence": 0.5},
                "gluttony": {"score": 0, "confidence": 0.5},
                "envy": {"score": 0, "confidence": 0.5},
                "sloth": {"score": 0, "confidence": 0.5}
            },
            "quality": {"score": 80, "tier": "high"}
        }
        profile_b = {
            "user_id": "b",
            "sins": {
                "wrath": {"score": -2.5, "confidence": 0.85},  # High virtue
                "greed": {"score": 0, "confidence": 0.5},
                "pride": {"score": 0, "confidence": 0.5},
                "lust": {"score": 0, "confidence": 0.5},
                "gluttony": {"score": 0, "confidence": 0.5},
                "envy": {"score": 0, "confidence": 0.5},
                "sloth": {"score": 0, "confidence": 0.5}
            },
            "quality": {"score": 80, "tier": "high"}
        }

        result = similarity_service.calculate_similarity(profile_a, profile_b)

        # Should find wrath overlap
        overlapping_sins = [b["sin"] for b in result["breakdown"]]
        assert "wrath" in overlapping_sins
    def test_opposite_directions_no_overlap(self, similarity_service):
        """One virtue, one vice on same trait should NOT create overlap."""
        profile_a = {
            "user_id": "a",
            "sins": {
                "wrath": {"score": -3.0, "confidence": 0.90},  # High virtue
                "greed": {"score": 0, "confidence": 0.5},
                "pride": {"score": 0, "confidence": 0.5},
                "lust": {"score": 0, "confidence": 0.5},
                "gluttony": {"score": 0, "confidence": 0.5},
                "envy": {"score": 0, "confidence": 0.5},
                "sloth": {"score": 0, "confidence": 0.5}
            },
            "quality": {"score": 80, "tier": "high"}
        }
        profile_b = {
            "user_id": "b",
            "sins": {
                "wrath": {"score": 3.0, "confidence": 0.88},  # High vice
(opposite!)
                "greed": {"score": 0, "confidence": 0.5},
                "pride": {"score": 0, "confidence": 0.5},
                "lust": {"score": 0, "confidence": 0.5},
                "gluttony": {"score": 0, "confidence": 0.5},
                "envy": {"score": 0, "confidence": 0.5},
                "sloth": {"score": 0, "confidence": 0.5}
            },
            "quality": {"score": 80, "tier": "high"}
        }

        result = similarity_service.calculate_similarity(profile_a, profile_b)

        # Should NOT find wrath overlap
        overlapping_sins = [b["sin"] for b in result["breakdown"]]
        assert "wrath" not in overlapping_sins

    def test_neutral_scores_no_overlap(self, similarity_service):
        """Neutral scores (near 0) should NOT create overlap."""
        profile_a = {
            "user_id": "a",
            "sins": {
                "greed": {"score": 0.3, "confidence": 0.60},  # Within neutral
threshold
                "pride": {"score": 0, "confidence": 0.5},
                "lust": {"score": 0, "confidence": 0.5},
                "wrath": {"score": 0, "confidence": 0.5},
                "gluttony": {"score": 0, "confidence": 0.5},
                "envy": {"score": 0, "confidence": 0.5},
                "sloth": {"score": 0, "confidence": 0.5}
            },
            "quality": {"score": 80, "tier": "high"}
        }
        profile_b = {
            "user_id": "b",
            "sins": {
                "greed": {"score": 0.4, "confidence": 0.55},  # Within neutral
threshold
                "pride": {"score": 0, "confidence": 0.5},
                "lust": {"score": 0, "confidence": 0.5},
                "wrath": {"score": 0, "confidence": 0.5},
                "gluttony": {"score": 0, "confidence": 0.5},
                "envy": {"score": 0, "confidence": 0.5},
                "sloth": {"score": 0, "confidence": 0.5}
            },
            "quality": {"score": 80, "tier": "high"}
        }

        result = similarity_service.calculate_similarity(profile_a, profile_b)

        # Neutral scores don't count as overlap
        overlapping_sins = [b["sin"] for b in result["breakdown"]]
        assert "greed" not in overlapping_sins


class TestSimilarityServiceWeights:
    """Tests for sin weight application."""

    def test_wrath_weight_highest(self, similarity_service):
        """Wrath overlap should contribute most (weight 1.5)."""
        # Profile with only wrath overlap
        profile_wrath = {
            "user_id": "wrath",
            "sins": {
                "wrath": {"score": -3.0, "confidence": 0.90},
                "greed": {"score": 0, "confidence": 0.5},
                "pride": {"score": 0, "confidence": 0.5},
                "lust": {"score": 0, "confidence": 0.5},
                "gluttony": {"score": 0, "confidence": 0.5},
                "envy": {"score": 0, "confidence": 0.5},
                "sloth": {"score": 0, "confidence": 0.5}
            },
            "quality": {"score": 80, "tier": "high"}
        }

        # Profile with only envy overlap (lowest weight 0.7)
        profile_envy = {
            "user_id": "envy",
            "sins": {
                "envy": {"score": -2.0, "confidence": 0.85},
                "wrath": {"score": 0, "confidence": 0.5},
                "greed": {"score": 0, "confidence": 0.5},
                "pride": {"score": 0, "confidence": 0.5},
                "lust": {"score": 0, "confidence": 0.5},
                "gluttony": {"score": 0, "confidence": 0.5},
                "sloth": {"score": 0, "confidence": 0.5}
            },
            "quality": {"score": 80, "tier": "high"}
        }

        # Partner with same traits
        partner = {
            "user_id": "partner",
            "sins": {
                "wrath": {"score": -2.5, "confidence": 0.88},
                "envy": {"score": -2.5, "confidence": 0.82},
                "greed": {"score": 0, "confidence": 0.5},
                "pride": {"score": 0, "confidence": 0.5},
                "lust": {"score": 0, "confidence": 0.5},
                "gluttony": {"score": 0, "confidence": 0.5},
                "sloth": {"score": 0, "confidence": 0.5}
            },
            "quality": {"score": 80, "tier": "high"}
        }

        result_wrath = similarity_service.calculate_similarity(profile_wrath,
partner)
        result_envy = similarity_service.calculate_similarity(profile_envy,
```

````python
partner)

        # Wrath overlap should produce higher score than envy overlap
        assert result_wrath["raw_score"] > result_envy["raw_score"]


class TestSimilarityServiceTiers:
    """Tests for similarity tier classification."""

    def test_tier_classification(self, similarity_service):
        """Similarity scores should map to correct tiers."""
        # Default threshold is 0.40

        # High similarity profile pair
        high_sim_a = {
            "user_id": "a",
            "sins": {
                "greed": {"score": -3.0, "confidence": 0.90},
                "pride": {"score": 2.5, "confidence": 0.85},
                "lust": {"score": 3.0, "confidence": 0.88},
                "wrath": {"score": -2.5, "confidence": 0.92},
                "gluttony": {"score": 2.0, "confidence": 0.80},
                "envy": {"score": -2.0, "confidence": 0.78},
                "sloth": {"score": 1.5, "confidence": 0.82}
            },
            "quality": {"score": 85, "tier": "high"}
        }
        high_sim_b = {
            "user_id": "b",
            "sins": {
                "greed": {"score": -2.5, "confidence": 0.88},
                "pride": {"score": 3.0, "confidence": 0.82},
                "lust": {"score": 2.5, "confidence": 0.85},
                "wrath": {"score": -3.0, "confidence": 0.90},
                "gluttony": {"score": 2.5, "confidence": 0.78},
                "envy": {"score": -2.5, "confidence": 0.80},
                "sloth": {"score": 2.0, "confidence": 0.85}
            },
            "quality": {"score": 82, "tier": "high"}
        }

        result = similarity_service.calculate_similarity(high_sim_a, high_sim_b)

        # Should be above threshold with good display mode
        assert result["adjusted_score"] >= 0.40
        assert result["tier"] in ["strong", "moderate"]
````

---

### 10.6 Integration Tests
````python
# tests/integration/test_pipeline.py

import pytest
from unittest.mock import AsyncMock, patch


class TestFullPipeline:
    """End-to-end pipeline integration tests."""

    @pytest.mark.asyncio
    async def test_question_to_similarity_pipeline(
        self,
````

```
        sample_question,
        sample_answer_cooperative,
        mock_gemini_service_factory,
        parsed_response_cooperative,
        profile_service,
        similarity_service
    ):
        """Test complete flow: question → parse → profile → similarity."""

        # Setup mock Gemini
        mock_gemini = mock_gemini_service_factory(parsed_response_cooperative)

        # Step 1: Parse 6 responses
        parsed_responses = []
        for i in range(6):
            result = await mock_gemini.parse_response(sample_question,
sample_answer_cooperative)
            parsed_responses.append(result)

        # Step 2: Aggregate into profile
        profile_a = profile_service.aggregate_profile(
            user_id="user_a",
            parsed_responses=parsed_responses
        )

        # Verify profile structure
        assert "sins" in profile_a
        assert all(sin in profile_a["sins"] for sin in
                   ["greed", "pride", "lust", "wrath", "gluttony", "envy",
"sloth"])
        assert profile_a["quality"]["tier"] != "rejected"

        # Step 3: Compare with another profile (use fixture)
        profile_b = {
            "user_id": "user_b",
            "sins": {
                "greed": {"score": -1.5, "confidence": 0.80},
                "pride": {"score": 0.8, "confidence": 0.65},
                "lust": {"score": 0.5, "confidence": 0.55},
                "wrath": {"score": -2.8, "confidence": 0.88},
                "gluttony": {"score": 0.2, "confidence": 0.50},
                "envy": {"score": -0.8, "confidence": 0.62},
                "sloth": {"score": 0.8, "confidence": 0.68}
            },
            "quality": {"score": 75, "tier": "moderate"}
        }

        # Step 4: Calculate similarity
        similarity = similarity_service.calculate_similarity(profile_a,
profile_b)

        # Verify similarity output
        assert "raw_score" in similarity
        assert "adjusted_score" in similarity
        assert "breakdown" in similarity
        assert "tier" in similarity
        assert 0 <= similarity["raw_score"] <= 1


class TestAPIEndpoints:
    """Integration tests for FastAPI endpoints."""

    @pytest.mark.asyncio
    async def test_submit_questionnaire_endpoint(self):
        """Test POST /api/questionnaire/submit endpoint."""
        from httpx import ASGITransport, AsyncClient
        from main import app

        async with AsyncClient(
            transport=ASGITransport(app=app),
            base_url="http://test"
        ) as client:
            response = await client.post(
                "/api/questionnaire/submit",
                json={
                    "user_id": "test_user_123",
                    "user_name": "Test User",
                    "responses": {
                        "q1": "I would split the bill evenly to keep things simple.",
                        "q2": "I prefer planning trips in advance with clear itineraries.",
                        "q3": "When someone disagrees with me, I try to understand their perspective.",
                        "q4": "I believe success comes from hard work and dedication.",
                        "q5": "I enjoy trying new experiences and meeting new people.",
                        "q6": "I think it's important to help others when you can."
                    }
                }
            )

            assert response.status_code == 200
            data = response.json()
            assert data["status"] in ["processing", "success"]
            assert "user_id" in data

    @pytest.mark.asyncio
    async def test_health_endpoint(self):
        """Test GET /health endpoint."""
        from httpx import ASGITransport, AsyncClient
        from main import app

        async with AsyncClient(
            transport=ASGITransport(app=app),
            base_url="http://test"
        ) as client:
            response = await client.get("/health")

            assert response.status_code == 200
            data = response.json()
            assert data["status"] == "healthy"
```

---

### 10.7 Input/Output Validation Criteria

**GeminiService.parse_response Input/Output:**

| Field | Input Constraints | Output Constraints |
|-------|-------------------|--------------------|
| question | String, non-empty | N/A |
| answer | String, ≥25 words recommended | N/A |
| sins.{sin}.score | N/A | Float, -5.0 to +5.0 |
| sins.{sin}.confidence | N/A | Float, 0.0 to 1.0 |

~~~
| sins.{sin}.evidence | N/A | String (may be empty) |
| word_count | N/A | Integer ≥ 0 |
| discrepancies | N/A | List of strings |

**Example Input:**
````python
question = "You and friends just finished dinner. Everyone contributed
differently. What's your approach?"
answer = "I usually suggest we split evenly because it keeps things simple and
nobody feels awkward. But if someone clearly ordered way less, I'd offer to
cover more of my share. Money isn't worth making friends uncomfortable over."
````

**Example Output:**
````python
{
    "sins": {
        "greed": {"score": -2.0, "confidence": 0.85, "evidence": "Money isn't
worth making friends uncomfortable"},
        "pride": {"score": 0.5, "confidence": 0.60, "evidence": ""},
        "lust": {"score": 0.0, "confidence": 0.50, "evidence": ""},
        "wrath": {"score": -3.0, "confidence": 0.90, "evidence": "Nobody feels
awkward"},
        "gluttony": {"score": 0.0, "confidence": 0.50, "evidence": ""},
        "envy": {"score": -1.0, "confidence": 0.65, "evidence": ""},
        "sloth": {"score": 1.0, "confidence": 0.70, "evidence": "Split evenly
because it keeps things simple"}
    },
    "liwc_signals": {
        "first_person_singular": 0.08,
        "first_person_plural": 0.02,
        "negative_emotion": 0.01,
        "positive_emotion": 0.02
    },
    "discrepancies": [],
    "word_count": 52
}
````

---

**ProfileService.aggregate_profile Input/Output:**

| Field | Input Constraints | Output Constraints |
|-------|-------------------|-------------------|
| user_id | String, non-empty | Same as input |
| parsed_responses | List of ≥4 parsed responses | N/A |
| sins.{sin}.score | N/A | Float, -5.0 to +5.0 |
| sins.{sin}.confidence | N/A | Float, 0.0 to 1.0 |
| sins.{sin}.variance | N/A | Float ≥ 0 |
| sins.{sin}.high_variance_flag | N/A | Boolean |
| quality.score | N/A | Float, 0 to 100 |
| quality.tier | N/A | "high" \| "moderate" \| "low" \| "rejected" |

**Example Input:**
````python
user_id = "user_123"
parsed_responses = [
    {"sins": {"greed": {"score": -2.0, "confidence": 0.85}, ...}, "word_count":
52},
    {"sins": {"greed": {"score": -1.5, "confidence": 0.80}, ...}, "word_count":
68},
    {"sins": {"greed": {"score": -2.5, "confidence": 0.88}, ...}, "word_count":
45},
~~~

````python
    {"sins": {"greed": {"score": -1.8, "confidence": 0.82}, ...}, "word_count": 71},
    {"sins": {"greed": {"score": -2.2, "confidence": 0.86}, ...}, "word_count": 55},
    {"sins": {"greed": {"score": -1.9, "confidence": 0.84}, ...}, "word_count": 62}
]
````

**Example Output:**
````python
{
    "user_id": "user_123",
    "version": 1,
    "created_at": "2026-01-29T14:30:00Z",
    "sins": {
        "greed": {
            "score": -2.0,
            "confidence": 0.84,
            "variance": 0.12,
            "item_count": 6,
            "method": "cwmv",
            "high_variance_flag": False
        },
        # ... other sins
    },
    "quality": {
        "score": 78.5,
        "tier": "moderate",
        "components": {
            "consistency": 82.0,
            "variance": 85.0,
            "style": 75.0,
            "engagement": 72.0
        }
    },
    "flags": []
}
````

---

**SimilarityService.calculate_similarity Input/Output:**

| Field | Input Constraints | Output Constraints |
|-------|-------------------|--------------------|
| profile_a | Valid profile with sins dict | N/A |
| profile_b | Valid profile with sins dict | N/A |
| raw_score | N/A | Float, 0.0 to 1.0 |
| adjusted_score | N/A | Float, 0.0 to 1.0 |
| breakdown | N/A | List of overlap details |
| tier | N/A | "strong" \| "moderate" \| "weak" \| "insufficient" |

**Example Input:**
````python
profile_a = {
    "user_id": "user_a",
    "sins": {
        "greed": {"score": -2.0, "confidence": 0.85},
        "pride": {"score": 0.5, "confidence": 0.60},
        "lust": {"score": 1.5, "confidence": 0.75},
        "wrath": {"score": -3.0, "confidence": 0.90},
        "gluttony": {"score": 0.8, "confidence": 0.62},
        "envy": {"score": -1.5, "confidence": 0.68},
        "sloth": {"score": 0.2, "confidence": 0.55}
    },
    "quality": {"score": 78, "tier": "moderate"}
}

profile_b = {
    "user_id": "user_b",
    "sins": {
        "greed": {"score": -2.5, "confidence": 0.82},
        "pride": {"score": 0.8, "confidence": 0.65},
        "lust": {"score": 2.0, "confidence": 0.78},
        "wrath": {"score": -2.5, "confidence": 0.88},
        "gluttony": {"score": 1.2, "confidence": 0.60},
        "envy": {"score": -2.0, "confidence": 0.72},
        "sloth": {"score": 0.5, "confidence": 0.58}
    },
    "quality": {"score": 81, "tier": "high"}
}
````

**Example Output:**
````python
{
    "raw_score": 0.52,
    "adjusted_score": 0.48,
    "quality_multiplier": 0.92,
    "breakdown": [
        {"sin": "greed", "direction": "virtue", "score_a": -2.0, "score_b":
-2.5, "contribution": 0.082},
        {"sin": "lust", "direction": "vice", "score_a": 1.5, "score_b": 2.0,
"contribution": 0.089},
        {"sin": "wrath", "direction": "virtue", "score_a": -3.0, "score_b":
-2.5, "contribution": 0.185},
        {"sin": "envy", "direction": "virtue", "score_a": -1.5, "score_b": -2.0,
"contribution": 0.063}
    ],
    "overlap_count": 4,
    "tier": "moderate",
    "display_mode": "full"
}
````

---

### 10.8 Edge Case Tests
````python
# tests/unit/test_edge_cases.py

import pytest


class TestEdgeCases:
    """Tests for edge cases and boundary conditions."""

    def test_empty_answer_handling(self, mock_gemini_service):
        """Empty answer should not crash, but return low confidence."""
        # This would be handled by validation before reaching Gemini
        pass

    def test_extremely_long_answer(self):
        """Answers exceeding 500 words should be truncated."""
        long_answer = "word " * 600  # 600 words
        # Should be truncated to ~500 words before processing
        truncated = " ".join(long_answer.split()[:500])
````

```
        assert len(truncated.split()) == 500

    def test_non_english_answer(self):
        """Non-English answers should return low confidence, not crash."""
        french_answer = "Je préfère partager l'addition également entre amis."
        # Gemini should still process, but LIWC signals will be minimal
        pass

    def test_all_neutral_scores(self, profile_service):
        """Profile with all neutral scores should still be valid."""
        neutral_sins = {
            sin: {"score": 0, "confidence": 0.5, "evidence": ""}
            for sin in ["greed", "pride", "lust", "wrath", "gluttony", "envy",
"sloth"]
        }

        parsed_responses = [
            {"sins": neutral_sins, "word_count": 50, "discrepancies": []}
            for _ in range(6)
        ]

        result = profile_service.aggregate_profile(
            user_id="neutral_user",
            parsed_responses=parsed_responses
        )

        # Should be valid but possibly flagged for MRS
        assert result["quality"]["tier"] != "rejected"

    def test_all_extreme_scores(self, profile_service):
        """Profile with all extreme scores should be flagged for ERS."""
        extreme_sins = {
            sin: {"score": 5 if i % 2 == 0 else -5, "confidence": 0.9,
"evidence": ""}
            for i, sin in enumerate(["greed", "pride", "lust", "wrath",
"gluttony", "envy", "sloth"])
        }

        parsed_responses = [
            {"sins": extreme_sins, "word_count": 80, "discrepancies": []}
            for _ in range(6)
        ]

        result = profile_service.aggregate_profile(
            user_id="extreme_user",
            parsed_responses=parsed_responses
        )

        # Should be flagged for ERS
        assert result["response_styles"]["ers"]["flag"] == True

    def test_rejected_profile_similarity(self, similarity_service):
        """Similarity with rejected profile should return insufficient
result."""
        rejected_profile = {
            "user_id": "rejected",
            "sins": None,
            "quality": {"score": 0, "tier": "rejected"}
        }
        valid_profile = {
            "user_id": "valid",
            "sins": {
                sin: {"score": 0, "confidence": 0.7}
                for sin in ["greed", "pride", "lust", "wrath", "gluttony",
"envy", "sloth"]
            },
            "quality": {"score": 75, "tier": "moderate"}
        }

        result = similarity_service.calculate_similarity(rejected_profile,
valid_profile)

        assert result["tier"] == "insufficient"

    def test_boundary_score_values(self):
        """Scores exactly at boundaries (-5, 0, +5) should be handled
correctly."""
        boundary_cases = [
            {"score": -5.0, "expected_direction": "virtue"},
            {"score": 0.0, "expected_direction": None},  # Neutral
            {"score": 5.0, "expected_direction": "vice"},
            {"score": -0.5, "expected_direction": None},  # Within neutral
threshold
            {"score": 0.5, "expected_direction": None},   # Within neutral
threshold
            {"score": -0.51, "expected_direction": "virtue"},  # Just outside
neutral
            {"score": 0.51, "expected_direction": "vice"}     # Just outside
neutral
        ]

        for case in boundary_cases:
            # Test that boundary values are classified correctly
            score = case["score"]
            if abs(score) <= 0.5:
                assert case["expected_direction"] is None
            elif score < -0.5:
                assert case["expected_direction"] == "virtue"
            else:
                assert case["expected_direction"] == "vice"
```

---

### 10.9 Running Tests

**Run all tests:**
```bash
pytest
```

**Run only unit tests:**
```bash
pytest -m unit
```

**Run with coverage report:**
```bash
pytest --cov=services --cov-report=html
```

**Run specific test file:**
```bash
pytest tests/unit/test_gemini_service.py -v
```

**Run tests matching pattern:**
```bash
```

````
pytest -k "test_similarity" -v
````

---

### 10.10 CI/CD Integration

**.github/workflows/test.yml:**
````yaml
name: Test Suite

on:
  push:
    branches: [main, develop]
  pull_request:
    branches: [main]

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v4

      - name: Set up Python
        uses: actions/setup-python@v5
        with:
          python-version: '3.11'

      - name: Install dependencies
        run: |
          pip install -r requirements.txt
          pip install -r requirements-test.txt

      - name: Run unit tests
        run: pytest -m unit -v --tb=short

      - name: Run integration tests
        run: pytest -m integration -v --tb=short
        env:
          USE_MOCK_GEMINI: "true"

      - name: Generate coverage report
        run: pytest --cov=services --cov-report=xml

      - name: Upload coverage
        uses: codecov/codecov-action@v4
        with:
          file: ./coverage.xml
````

---

### 10.11 Summary

**Test Coverage Goals:**

| Component | Target Coverage | Critical Paths |
|-----------|-----------------|----------------|
| GeminiService | ≥ 85% | Validation, JSON extraction, LIWC, discrepancies |
| ProfileService | ≥ 90% | Aggregation, quality scoring, response styles |
| SimilarityService | ≥ 95% | Positive overlap, weights, tiers |
| API Endpoints | ≥ 80% | Submit, health, compare |

**Key Validation Rules:**

| Rule | Location | Enforcement |
|------|----------|-------------|
| Score range [-5, +5] | GeminiService | Raises ValueError |
| Confidence range [0, 1] | GeminiService | Raises ValueError |
| All 7 sins present | GeminiService | Raises ValueError |
| Minimum 4 valid responses | ProfileService | Returns rejected profile |
| Quality tier calculation | ProfileService | Composite score formula |
| Positive overlap only | SimilarityService | Direction check |
| Neutral threshold (±0.5) | SimilarityService | No overlap contribution |

**What This Section Achieves:**

1. Complete test fixtures for all services
2. Unit tests covering all validation logic
3. Integration tests for end-to-end pipeline
4. Input/output specifications with examples
5. Edge case handling documentation
6. CI/CD configuration for automated testing

---

Ready for Section 11 (Appendices) or final compilation?

## Section 11: Appendices

---

### Appendix A: Complete Prompt Templates

#### A.1 Single-Trait Parsing Prompt

This is the production prompt for parsing one trait at a time (used when running 7 parallel calls per question):

python
```
SINGLE_TRAIT_PROMPT_TEMPLATE = """Analyze this response for {sin_upper} signals only.

{sin_name} Scale (-5 to +5):
-5 = {anchor_low}
 0 = Neutral: No clear signal either direction
+5 = {anchor_high}

Question: "{question}"
Response: "{answer}"

Scoring Guidelines:
- Score 0 if the response gives no indication of this trait
- Score -1 to -2 for mild virtue signals, -3 to -5 for strong virtue
- Score +1 to +2 for mild vice signals, +3 to +5 for strong vice
- Confidence reflects how clearly the text supports your score
- Evidence must be a direct quote or empty string if no clear signal

Return ONLY valid JSON (no markdown, no preamble):
{{"score": <-5 to +5>, "confidence": <0.0 to 1.0>, "evidence": "<direct quote or empty>"}}"""
```

#### A.2 Trait Anchor Definitions

python
```
TRAIT_ANCHORS = {
    "greed": {
        "low": "Extremely generous, prioritises others' needs, unconcerned with personal gain",
        "high": "Highly materialistic, self-interested, resource-hoarding, transactional"
    },
    "pride": {
        "low": "Very humble, self-deprecating, avoids spotlight, downplays achievements",
        "high": "Status-seeking, ego-driven, needs validation, self-promotional"
    },
    "lust": {
        "low": "Extremely cautious, risk-averse, prefers routine, resists novelty",
        "high": "Highly impulsive, novelty-seeking, acts on urges, spontaneous"
    },
    "wrath": {
        "low": "Extreme conflict avoidance, never expresses anger, highly accommodating",
        "high": "Quick to anger, confrontational, expresses frustration readily"
    },
    "gluttony": {
        "low": "Highly moderate, practices strict self-control, ascetic tendencies",
        "high": "Strongly indulgent, struggles with restraint, excess-seeking"
    },
    "envy": {
        "low": "Deeply content, never compares self to others, secure in own position",
        "high": "Constantly competitive, resentful of others' success, comparison-driven"
    },
    "sloth": {
        "low": "Extremely proactive, takes initiative constantly, high energy",
        "high": "Avoidant, passive, procrastinates, takes path of least resistance"
    }
}
```

### A.3 All-Trait Parsing Prompt (Alternative)

This prompt parses all 7 traits in a single call (used when minimising API calls):

python
```
ALL_TRAIT_PROMPT_TEMPLATE = """Role: Expert Psychological Profiler using the Seven Deadly Sins framework.
Task: Analyze this questionnaire response and score each trait from -5 (extreme virtue) to +5 (extreme vice).

Question: "{question}"
Response: "{answer}"

TRAIT DEFINITIONS:
- GREED: Material self-interest vs. generosity (-5 = selfless, +5 = materialistic)
- PRIDE: Status-seeking vs. humility (-5 = humble, +5 = ego-driven)
- LUST: Impulsivity vs. restraint (-5 = cautious, +5 = spontaneous)
- WRATH: Confrontation vs. accommodation (-5 = conflict-avoidant, +5 = confrontational)
- GLUTTONY: Indulgence vs. moderation (-5 = ascetic, +5 = indulgent)
- ENVY: Comparison vs. contentment (-5 = secure, +5 = competitive)
- SLOTH: Passivity vs. initiative (-5 = proactive, +5 = avoidant)
```

```
SCORING RULES:
- Score 0 = No clear signal (neutral)
- Negative scores = Virtue direction
- Positive scores = Vice direction
- Confidence (0.0-1.0) = How clearly the text supports your score
- Evidence = Direct quote from response, or empty if no clear signal

Return ONLY valid JSON:
{{
  "greed": {{"score": 0, "confidence": 0.5, "evidence": ""}},
  "pride": {{"score": 0, "confidence": 0.5, "evidence": ""}},
  "lust": {{"score": 0, "confidence": 0.5, "evidence": ""}},
  "wrath": {{"score": 0, "confidence": 0.5, "evidence": ""}},
  "gluttony": {{"score": 0, "confidence": 0.5, "evidence": ""}},
  "envy": {{"score": 0, "confidence": 0.5, "evidence": ""}},
  "sloth": {{"score": 0, "confidence": 0.5, "evidence": ""}}
}}

Be nuanced — most responses show mixed signals. Only use extreme scores (±4, ±5) for very clear signals."""
```

### A.4 Gemini Configuration

python

```python
GEMINI_CONFIG = {
    "model_chain": [
        "gemini-3-flash-preview",   # Primary: fast, cheap
        "gemini-3-pro-preview",     # Fallback: more capable
        "gemini-2.5-flash"          # Final fallback: stable
    ],

    "safety_settings": [
        {"category": "HARM_CATEGORY_HARASSMENT", "threshold": "BLOCK_NONE"},
        {"category": "HARM_CATEGORY_HATE_SPEECH", "threshold": "BLOCK_NONE"},
        {"category": "HARM_CATEGORY_SEXUALLY_EXPLICIT", "threshold": "BLOCK_NONE"},
        {"category": "HARM_CATEGORY_DANGEROUS_CONTENT", "threshold":
"BLOCK_NONE"}
    ],

    "generation_config": {
        "temperature": 0.3,          # Low for consistency
        "top_p": 0.95,
        "top_k": 40,
        "max_output_tokens": 1024,
        "response_mime_type": "application/json"
    },

    "timeout_seconds": 60,
    "max_retries": 3,
    "retry_base_delay": 1.0,
    "retry_max_delay": 30.0
}
```

---

### Appendix B: Felix's PIIP Questions

#### B.1 Question Specifications

| ID | Short Name | Primary Targets | Secondary Targets |
|----|------------|-----------------|-------------------|
| Q1 | Group Dinner Check | greed, wrath | pride, sloth |
| Q2 | Unexpected Expense | greed, sloth | wrath, gluttony |
| Q3 | Weekend Off | sloth, lust, gluttony | pride |
| Q4 | Unequal Split | wrath, pride | envy, sloth |
| Q5 | Friend Crisis | lust, wrath | pride, greed |
| Q6 | Feedback Received | pride, wrath | envy |

#### B.2 Full Question Text

**Q1: The Group Dinner Check**
```
The bill arrives at a group dinner. Everyone contributed differently to the total. What's your approach?
```

*Design rationale:* Elicits attitudes toward fairness (greed), willingness to create conflict (wrath), and preference for simple solutions (sloth).

---

**Q2: The Unexpected Expense**
```
Your car needs a $1,200 repair you didn't budget for. Walk me through your thought process.
```

*Design rationale:* Reveals financial attitudes (greed), problem-solving approach (sloth), and stress response (wrath).

---

**Q3: The Weekend Off**
```
You have a completely free weekend with no obligations. How do you spend it?
```

*Design rationale:* Shows energy level and initiative (sloth), novelty-seeking (lust), and indulgence patterns (gluttony).

---

**Q4: The Unequal Split**
```
You're working on a group project where one person is contributing significantly less than everyone else. How do you handle this?
```

*Design rationale:* Tests conflict tolerance (wrath), fairness concerns (envy), and leadership/ego (pride).

---

**Q5: The Friend Crisis**
```
Your best friend calls with an emergency the same night as a planned date you've been looking forward to. What's your thinking process?
```

*Design rationale:* Reveals impulse vs. commitment balance (lust), interpersonal priorities (greed), and decision-making under pressure (wrath).

---

**Q6: The Feedback Received**
```
Someone you respect gives you critical feedback about a blind spot they've noticed in your behavior. How do you process and respond to this?
```

*Design rationale:* Tests ego response (pride), defensiveness (wrath), and comparison tendencies (envy).

---

#### B.3 Response Requirements

| Requirement | Value | Enforcement |
|-------------|-------|-------------|
| Minimum words | 25 | Soft validation (warning) |
| Recommended words | 50-150 | UI guidance |
| Maximum words | 250 | Hard limit (truncation) |
| Required questions | 4 of 6 | Profile rejection below |

#### B.4 Example Responses

**Q1 — Low Wrath, Low Greed Response:**
```
I usually suggest we split evenly because it keeps things simple and nobody
feels awkward. But if someone clearly ordered way less, I'd offer to cover
more of my share. Money isn't worth making friends uncomfortable over. I'd
rather pay a bit extra than have anyone feel like they're being taken
advantage of or create tension in the group.
```

*Expected parsing:* greed ≈ -2, wrath ≈ -3, sloth ≈ +1

---

**Q1 — High Wrath, High Greed Response:**
```
I always track what I ordered and pay exactly that. Why should I subsidize
someone else's expensive steak when I had a salad? I'm not afraid to speak
up about it either — it's about fairness. If people get uncomfortable,
that's their problem. I've worked hard for my money and I'm not going to
let social pressure make me overpay.
```

*Expected parsing:* greed ≈ +2.5, wrath ≈ +2.5, pride ≈ +3

---

**Q4 — High Wrath, High Pride Response:**
```
```

~~~~
I'd call them out directly in the next meeting. Everyone deserves to know
who's actually contributing and who's coasting. I've done this before and
yes, it's uncomfortable, but someone has to have the guts to say something.
I usually end up leading these conversations because nobody else will.
```

*Expected parsing:* wrath ≈ +3, pride ≈ +3.5, envy ≈ +1

---

**Q4 — Low Wrath, Low Pride Response:**
```
I'd probably try to understand why they're not contributing — maybe they're
dealing with something. I might pick up some of their slack quietly rather
than making a big deal. It's not worth damaging the relationship or making

the project awkward. We all have off periods.
~~~~

*Expected parsing:* wrath ≈ -3, pride ≈ -1, sloth ≈ +1.5

---

## Appendix C: Data Schema Reference

### C.1 Parsed Response Schema

json

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "ParsedResponse",
  "description": "Output from GeminiService.parse_response()",
  "type": "object",
  "required": ["sins", "word_count"],
  "properties": {
    "sins": {
      "type": "object",
      "required": ["greed", "pride", "lust", "wrath", "gluttony", "envy", "sloth"],
      "additionalProperties": false,
      "properties": {
        "greed": { "$ref": "#/definitions/SinScore" },
        "pride": { "$ref": "#/definitions/SinScore" },
        "lust": { "$ref": "#/definitions/SinScore" },
        "wrath": { "$ref": "#/definitions/SinScore" },
        "gluttony": { "$ref": "#/definitions/SinScore" },
        "envy": { "$ref": "#/definitions/SinScore" },
        "sloth": { "$ref": "#/definitions/SinScore" }
```

```
      }
    },
    "liwc_signals": {
      "type": "object",
      "properties": {
        "first_person_singular": { "type": "number", "minimum": 0, "maximum": 1 },
        "first_person_plural": { "type": "number", "minimum": 0, "maximum": 1 },
        "negative_emotion": { "type": "number", "minimum": 0, "maximum": 1 },
        "positive_emotion": { "type": "number", "minimum": 0, "maximum": 1 },
        "certainty": { "type": "number", "minimum": 0, "maximum": 1 },
        "hedging": { "type": "number", "minimum": 0, "maximum": 1 }
      }
    },
    "discrepancies": {
      "type": "array",
      "items": { "type": "string" }
    },
    "word_count": {
      "type": "integer",
      "minimum": 0
    }
  },
  "definitions": {
    "SinScore": {
      "type": "object",
      "required": ["score", "confidence"],
      "properties": {
        "score": {
          "type": "number",
          "minimum": -5,
          "maximum": 5
        },
        "confidence": {
          "type": "number",
          "minimum": 0,
          "maximum": 1
        },
        "evidence": {
          "type": "string"
        }
      }
    }
  }
}
```

### C.2 User Profile Schema

json

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "UserProfile",
  "description": "Aggregated personality profile from ProfileService",
  "type": "object",
  "required": ["user_id", "version", "created_at", "sins", "quality"],
  "properties": {
    "user_id": { "type": "string" },
    "version": { "type": "integer", "minimum": 1 },
    "created_at": { "type": "string", "format": "date-time" },
    "updated_at": { "type": "string", "format": "date-time" },
    "sins": {
      "type": "object",
      "required": ["greed", "pride", "lust", "wrath", "gluttony", "envy", "sloth"],
      "additionalProperties": false,
      "properties": {
        "greed": { "$ref": "#/definitions/AggregatedSin" },
        "pride": { "$ref": "#/definitions/AggregatedSin" },
        "lust": { "$ref": "#/definitions/AggregatedSin" },
        "wrath": { "$ref": "#/definitions/AggregatedSin" },
        "gluttony": { "$ref": "#/definitions/AggregatedSin" },
        "envy": { "$ref": "#/definitions/AggregatedSin" },
        "sloth": { "$ref": "#/definitions/AggregatedSin" }
      }
    },
    "quality": { "$ref": "#/definitions/QualityScore" },
    "response_styles": { "$ref": "#/definitions/ResponseStyles" },
    "flags": {
      "type": "array",
      "items": { "type": "string" }
    },
    "metadata": {
      "type": "object",
      "properties": {
        "questions_answered": { "type": "integer" },
        "questions_valid": { "type": "integer" },
        "total_word_count": { "type": "integer" },
        "response_time_seconds": { "type": "number" },
        "gemini_model": { "type": "string" },
        "parsing_version": { "type": "string" }
      }
    }
```

```json
},
"definitions": {
  "AggregatedSin": {
    "type": "object",
    "required": ["score", "confidence"],
    "properties": {
      "score": { "type": "number", "minimum": -5, "maximum": 5 },
      "confidence": { "type": "number", "minimum": 0, "maximum": 1 },
      "variance": { "type": "number", "minimum": 0 },
      "item_count": { "type": "integer", "minimum": 0 },
      "method": { "type": "string", "enum": ["cwmv", "mean", "none"] },
      "high_variance_flag": { "type": "boolean" }
    }
  },
  "QualityScore": {
    "type": "object",
    "required": ["score", "tier"],
    "properties": {
      "score": { "type": "number", "minimum": 0, "maximum": 100 },
      "tier": { "type": "string", "enum": ["high", "moderate", "low", "rejected"] },
      "components": {
        "type": "object",
        "properties": {
          "consistency": { "type": "number" },
          "variance": { "type": "number" },
          "style": { "type": "number" },
          "engagement": { "type": "number" }
        }
      },
      "recommendation": { "type": "string" }
    }
  },
  "ResponseStyles": {
    "type": "object",
    "properties": {
      "ers": {
        "type": "object",
        "properties": {
          "flag": { "type": "boolean" },
          "percentage": { "type": "number" }
        }
      },
      "mrs": {
        "type": "object",
          "properties": {
            "flag": { "type": "boolean" },
            "percentage": { "type": "number" },
            "fast_completion": { "type": "boolean" }
          }
        },
        "patterns": {
          "type": "object",
          "properties": {
            "flag": { "type": "boolean" },
            "patterns": { "type": "array" }
          }
        }
      }
    }
  }
}
```

### C.3 Similarity Result Schema

json

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "SimilarityResult",
  "description": "Output from SimilarityService.calculate_similarity()",
  "type": "object",
  "required": ["raw_score", "adjusted_score", "overlap_count", "tier"],
  "properties": {
    "raw_score": {
      "type": "number",
      "minimum": 0,
      "maximum": 1
    },
    "adjusted_score": {
      "type": "number",
      "minimum": 0,
      "maximum": 1
    },
    "quality_multiplier": {
      "type": "number",
      "minimum": 0,
      "maximum": 1
    },
    "breakdown": {
        "type": "array",
        "items": {
          "type": "object",
          "properties": {
            "sin": { "type": "string" },
            "direction": { "type": "string", "enum": ["virtue", "vice"] },
            "score_a": { "type": "number" },
            "score_b": { "type": "number" },
            "trait_similarity": { "type": "number" },
            "avg_confidence": { "type": "number" },
            "weight": { "type": "number" },
            "contribution": { "type": "number" }
          }
        }
      },
      "overlap_count": {
        "type": "integer",
        "minimum": 0,
        "maximum": 7
      },
      "tier": {
        "type": "string",
        "enum": ["strong", "moderate", "weak", "insufficient"]
      },
      "display_mode": {
        "type": "string",
        "enum": ["full", "limited", "hidden"]
      },
      "quality_tier": {
        "type": "string"
      }
    }
}
```

**C.4 Pydantic Models (Python)**

python

```python
from pydantic import BaseModel, Field, validator
from typing import Optional, List, Dict
from datetime import datetime
from enum import Enum


class QualityTier(str, Enum):
    HIGH = "high"
    MODERATE = "moderate"
    LOW = "low"
    REJECTED = "rejected"


class AggregationMethod(str, Enum):
    CWMV = "cwmv"
    MEAN = "mean"
    NONE = "none"


class SimilarityTier(str, Enum):
    STRONG = "strong"
    MODERATE = "moderate"
    WEAK = "weak"
    INSUFFICIENT = "insufficient"


class SinScore(BaseModel):
    score: float = Field(..., ge=-5, le=5)
    confidence: float = Field(..., ge=0, le=1)
    evidence: str = ""

    @validator('score')
    def round_score(cls, v):
        return round(v, 2)


class AggregatedSin(BaseModel):
    score: float = Field(..., ge=-5, le=5)
    confidence: float = Field(..., ge=0, le=1)
    variance: float = Field(default=0, ge=0)
    item_count: int = Field(default=0, ge=0)
    method: AggregationMethod = AggregationMethod.NONE
    high_variance_flag: bool = False


class QualityComponents(BaseModel):
    consistency: float
    variance: float
    style: float
    engagement: float
```

```python
class QualityScore(BaseModel):
    score: float = Field(..., ge=0, le=100)
    tier: QualityTier
    components: Optional[QualityComponents] = None
    recommendation: Optional[str] = None


class ERSFlags(BaseModel):
    flag: bool = False
    percentage: float = 0


class MRSFlags(BaseModel):
    flag: bool = False
    percentage: float = 0
    fast_completion: bool = False


class PatternFlags(BaseModel):
    flag: bool = False
    patterns: List[Dict] = []


class ResponseStyles(BaseModel):
    ers: ERSFlags = ERSFlags()
    mrs: MRSFlags = MRSFlags()
    patterns: PatternFlags = PatternFlags()


class ProfileMetadata(BaseModel):
    questions_answered: int
    questions_valid: int
    total_word_count: int
    response_time_seconds: Optional[float] = None
    gemini_model: str
    parsing_version: str = "1.0.0"


class SinsProfile(BaseModel):
    greed: AggregatedSin
    pride: AggregatedSin
    lust: AggregatedSin
    wrath: AggregatedSin
    gluttony: AggregatedSin
    envy: AggregatedSin
    sloth: AggregatedSin


class UserProfile(BaseModel):
    user_id: str
    version: int = 1
    created_at: datetime
    updated_at: Optional[datetime] = None
    sins: SinsProfile
    quality: QualityScore
    response_styles: Optional[ResponseStyles] = None
    flags: List[str] = []
    metadata: Optional[ProfileMetadata] = None


class OverlapDetail(BaseModel):
    sin: str
    direction: str
    score_a: float
    score_b: float
    trait_similarity: float
    avg_confidence: float
    weight: float
    contribution: float


class SimilarityResult(BaseModel):
    raw_score: float = Field(..., ge=0, le=1)
    adjusted_score: float = Field(..., ge=0, le=1)
    quality_multiplier: float = Field(default=1.0, ge=0, le=1)
    breakdown: List[OverlapDetail] = []
    overlap_count: int = Field(..., ge=0, le=7)
    tier: SimilarityTier
    display_mode: str = "full"

    quality_tier: Optional[str] = None
```

---

## Appendix D: API Endpoint Reference

### D.1 Health Check

**GET /health**

Liveness check — returns 200 if application is running. Does NOT check Gemini connectivity.

**Response 200:**

json
```json
{
  "status": "healthy",
  "startup_time": "2026-01-29T14:30:00Z",
  "checks": {
    "app_initialized": true,
    "gemini_configured": true
  }
}
```

**Example:**

bash
```bash
curl http://localhost:8000/health
```

---

**GET /health/deep**

Deep health check — actually calls Gemini API. Takes 2-3 seconds. For manual monitoring only, NOT for Railway health checks.

**Response 200:**

json
```json
{
  "status": "healthy",
  "checks": {
    "app": "healthy",
    "gemini_api": "healthy"
  }
}
```

**Response 200 (degraded):**

json
```json
{
  "status": "degraded",
  "checks": {
```

~~~
    "app": "healthy",
    "gemini_api": "unhealthy: 503 Service Unavailable"
  }
}
```

---

#### D.2 Submit Questionnaire

**POST /api/questionnaire/submit**

Submit completed questionnaire for background processing.

**Request Headers:**
```

Content-Type: application/json
~~~

**Request Body:**

json
```json
{
  "user_id": "string (required)",
  "user_name": "string (required)",
  "responses": {
    "q1": "string (25-250 words)",
    "q2": "string (25-250 words)",
    "q3": "string (25-250 words)",
    "q4": "string (25-250 words)",
    "q5": "string (25-250 words)",
    "q6": "string (25-250 words)"
  }
}
```

**Response 200:**

json
```json
{
  "status": "processing",
  "user_id": "user_123",
  "message": "Questionnaire received and processing started",
  "estimated_time_seconds": 45
}
```

**Response 400:**

json
```json
{
  "detail": "Minimum 4 responses required"
}
```

**Response 400:**

json
```json
{
  "detail": "Response for q1 below minimum word count (25)"
}
```

**Example:**

bash
```bash
curl -X POST http://localhost:8000/api/questionnaire/submit \
  -H "Content-Type: application/json" \
  -d '{
    "user_id": "user_123",
    "user_name": "Alice",
    "responses": {
      "q1": "I usually suggest we split evenly because it keeps things simple...",
      "q2": "First I would check my emergency fund to see if I can cover it...",
      "q3": "Honestly, I would probably sleep in and then decide spontaneously...",
      "q4": "I would try to understand why they are not contributing first...",
      "q5": "This is tough. I think I would call my date and explain...",
      "q6": "I would take some time to process before responding..."
    }
  }'
```

---

**D.3 Get Profile**

**GET /api/profile/{user_id}**

Retrieve processed personality profile.

**Path Parameters:**

- `user_id` (string, required): The user's unique identifier

**Response 200:**

json
```json
{
  "user_id": "user_123",
  "version": 1,
  "created_at": "2026-01-29T14:30:00Z",
  "sins": {
    "greed": {"score": -1.8, "confidence": 0.82, "variance": 1.2, "high_variance_flag": false},
    "pride": {"score": 0.5, "confidence": 0.65, "variance": 2.1, "high_variance_flag": false},
    "lust": {"score": 1.2, "confidence": 0.75, "variance": 1.5, "high_variance_flag": false},
    "wrath": {"score": -2.5, "confidence": 0.88, "variance": 0.9, "high_variance_flag": false},
    "gluttony": {"score": 0.8, "confidence": 0.62, "variance": 2.5, "high_variance_flag": false},
    "envy": {"score": -0.8, "confidence": 0.70, "variance": 1.3, "high_variance_flag": false},
    "sloth": {"score": 0.3, "confidence": 0.68, "variance": 1.8, "high_variance_flag": false}
  },
  "quality": {
    "score": 78.5,
    "tier": "moderate"
  }
}
```

**Response 404:**

json
```json
{
  "detail": "Profile not found for user_id: user_123"
}
```

**Response 202 (still processing):**

json
```json
{
  "status": "processing",
  "user_id": "user_123",
  "progress": "Analyzing response 4 of 6"
}
```

---

**D.4 Calculate Similarity**

**POST /api/similarity/calculate**

Calculate perceived similarity between two profiles.

**Request Body:**

json
```
{
  "user_id_a": "string (required)",
  "user_id_b": "string (required)"
}
```

**Response 200:**

json
```
{
  "raw_score": 0.52,
  "adjusted_score": 0.48,
  "quality_multiplier": 0.92,
  "breakdown": [
    {"sin": "greed", "direction": "virtue", "contribution": 0.082},
    {"sin": "wrath", "direction": "virtue", "contribution": 0.185},
    {"sin": "lust", "direction": "vice", "contribution": 0.089}
  ],
  "overlap_count": 3,
  "tier": "moderate",
  "display_mode": "full"
}
```

**Response 400:**

json
```
{
  "detail": "Profile for user_id_a not found"
}
```

**Response 200 (insufficient data):**

json
```
{
  "raw_score": 0,
  "adjusted_score": 0,
  "breakdown": [],
  "overlap_count": 0,
  "tier": "insufficient",
  "display_mode": "hidden",
  "reason": "user_id_a profile quality tier is 'rejected'"
}
```

---

**D.5 Get Processing Status**

**GET /api/questionnaire/status/{user_id}**

Check processing status for a submitted questionnaire.

**Response 200 (processing):**

json
```
{
  "status": "processing",
  "user_id": "user_123",
  "stage": "parsing",
  "progress_percent": 65,
  "message": "Analyzing response 4 of 6"
}
```

**Response 200 (complete):**

json
```
{
  "status": "complete",
  "user_id": "user_123",
  "profile_ready": true,
  "quality_tier": "high"
}
```

**Response 200 (failed):**

json
```
{
  "status": "failed",
  "user_id": "user_123",
  "error": "Gemini API unavailable after 3 retries"
}
```
```` ``` ````

---
````

### Appendix E: Configuration Reference

#### E.1 Environment Variables

| Variable | Type | Required | Default | Description |
|----------|------|----------|---------|-------------|
| `GEMINI_API_KEY` | string | ✅ | - | Google AI API key (seal in Railway) |
| `RAILWAY_DEPLOYMENT_DRAINING_SECONDS` | int | ✅ | 120 | Grace period for in-flight requests during redeploy |
| `PORT` | int | Auto | 8000 | Railway-injected port |
| `ENVIRONMENT` | string | ❌ | production | deployment environment |
| `LOG_LEVEL` | string | ❌ | INFO | Logging verbosity (DEBUG, INFO, WARNING, ERROR) |
| `GEMINI_MODEL_PRIMARY` | string | ❌ | gemini-3-flash-preview | Primary Gemini model |
| `GEMINI_MODEL_FALLBACK` | string | ❌ | gemini-3-pro-preview | Fallback Gemini model |
| `GEMINI_TIMEOUT_SECONDS` | int | ❌ | 60 | Gemini API call timeout |
| `GEMINI_MAX_RETRIES` | int | ❌ | 3 | Maximum retry attempts |
| `REQUEST_TIMEOUT_SECONDS` | int | ❌ | 70 | FastAPI request timeout |
| `VISUAL_WEIGHT` | float | ❌ | 0.60 | Visual component weight (Stage 1) |
| `PERSONALITY_WEIGHT` | float | ❌ | 0.30 | Personality component weight (Stage 2) |
| `HLA_WEIGHT` | float | ❌ | 0.10 | Genetic component weight (Stage 3) |
| `STAGE2_THRESHOLD` | float | ❌ | 0.40 | Minimum similarity to proceed |

#### E.2 Application Constants

| Constant | Value | Location | Description |
|----------|-------|----------|-------------|
| `NEUTRAL_THRESHOLD` | 0.5 | SimilarityService | Scores within ±0.5 are neutral |
| `HIGH_VARIANCE_THRESHOLD` | 3.0 | ProfileService | SD above this triggers flag |
| `MIN_VALID_QUESTIONS` | 4 | ProfileService | Minimum for non-rejected profile |
| `MIN_WORD_COUNT` | 25 | Validation | Minimum words per response |
| `MAX_WORD_COUNT` | 250 | Validation | Maximum words per response |
| `ERS_THRESHOLD` | 0.40 | ProfileService | >40% extreme scores = ERS flag |
| `MRS_THRESHOLD` | 0.50 | ProfileService | >50% midpoint scores = MRS flag |
| `QUALITY_HIGH_THRESHOLD` | 80 | ProfileService | Quality score ≥80 = "high" tier |
| `QUALITY_LOW_THRESHOLD` | 60 | ProfileService | Quality score <60 = "low" tier |
| `MEDIAN_RESPONSE_TIME` | 300 | ProfileService | Expected completion time (seconds) |

#### E.3 Sin Weights

| Sin | Weight | Normalised |
|-----|--------|------------|
| wrath | 1.5 | 0.203 |
| sloth | 1.3 | 0.176 |

| | | |
|---|---|---|
| pride | 1.2 | 0.162 |
| lust | 1.0 | 0.135 |
| greed | 0.9 | 0.122 |
| gluttony | 0.8 | 0.108 |
| envy | 0.7 | 0.095 |
| **Total** | **7.4** | **1.000** |

---

### Appendix F: Worked Examples

#### F.1 User A: "The Peacemaker"

**Questionnaire Responses:**

**Q1 (Group Dinner Check):**
> I usually suggest we split evenly because it keeps things simple and nobody feels awkward. But if someone clearly ordered way less, I'd offer to cover more of my share. Money isn't worth making friends uncomfortable over. I'd rather pay a bit extra than have anyone feel like they're being taken advantage of or create tension in the group.

**Q2 (Unexpected Expense):**
> First, I'd take a breath and not panic. I'd check if I have enough in savings — I try to keep an emergency fund for exactly this. If not, I'd see if I could put it on a card and pay it off over a couple months. I wouldn't stress too much because cars need repairs sometimes, that's just life.

**Q3 (Weekend Off):**
> Honestly, I'd probably sleep in, make a nice breakfast, and see how I feel. Maybe meet up with friends if they're free, or just have a quiet day reading and watching shows. I don't need to fill every moment with activity. Sometimes doing nothing is exactly what you need.

**Q4 (Unequal Split):**
> I'd try to understand why first — maybe they're dealing with something. I might pick up some slack quietly rather than calling them out. If it continued, I'd have a gentle conversation about how we could redistribute work. I don't want to embarrass anyone or damage the relationship.

**Q5 (Friend Crisis):**
> My friend comes first. A date can be rescheduled but a friend in crisis needs me now. I'd text my date, explain honestly what happened, and hope they understand. If they don't, that tells me something about them anyway. Real emergencies don't wait for convenient timing.

**Q6 (Feedback Received):**

> I'd thank them for telling me — it takes courage to give honest feedback. I'd probably need some time to process it before responding fully. I'd think about whether there's truth to it and try not to get defensive. Growth comes from hearing things we don't always want to hear.

---

**Gemini Parsing Results:**

| Question | greed | pride | lust | wrath | gluttony | envy | sloth |
|----------|-------|-------|------|-------|----------|------|-------|
| Q1 | -2.0 (0.85) | 0.5 (0.60) | 0.0 (0.50) | -3.0 (0.90) | 0.0 (0.50) | -1.0 (0.65) | 1.0 (0.70) |
| Q2 | -1.5 (0.78) | -0.5 (0.55) | -1.0 (0.65) | -2.0 (0.80) | 0.0 (0.50) | 0.0 (0.50) | 0.5 (0.60) |
| Q3 | 0.0 (0.50) | -1.0 (0.60) | 0.5 (0.55) | 0.0 (0.50) | 1.5 (0.70) | 0.0 (0.50) | 2.0 (0.80) |
| Q4 | -1.0 (0.70) | -1.5 (0.72) | 0.0 (0.50) | -3.5 (0.92) | 0.0 (0.50) | -0.5 (0.55) | 1.0 (0.65) |
| Q5 | -2.5 (0.88) | 0.0 (0.50) | -1.5 (0.75) | -1.0 (0.68) | 0.0 (0.50) | 0.0 (0.50) | 0.0 (0.50) |
| Q6 | 0.0 (0.50) | -2.0 (0.82) | 0.0 (0.50) | -2.5 (0.85) | 0.0 (0.50) | -1.0 (0.60) | 0.0 (0.50) |

*Format: score (confidence)*

---

**Aggregation (Wrath example):**

Scores: [-3.0, -2.0, 0.0, -3.5, -1.0, -2.5]
Confidences: [0.90, 0.80, 0.50, 0.92, 0.68, 0.85]

Confidence SD = 0.16 (> 0.10, so use CWMV)

CWMV calculation:
```
Weighted sum = (-3.0×0.90) + (-2.0×0.80) + (0.0×0.50) + (-3.5×0.92) + (-1.0×0.68) + (-2.5×0.85)
             = -2.70 + -1.60 + 0.00 + -3.22 + -0.68 + -2.125
             = -10.325

Confidence sum = 0.90 + 0.80 + 0.50 + 0.92 + 0.68 + 0.85 = 4.65

Aggregated score = -10.325 / 4.65 = -2.22

Variance = 1.56 (SD = 1.25, below 3.0 threshold)
High variance flag = False
```

---

**Final Profile for User A:**

json

```
{
  "user_id": "user_a",
  "version": 1,
  "created_at": "2026-01-29T14:30:00Z",
  "sins": {
    "greed": {"score": -1.35, "confidence": 0.72, "variance": 0.82, "high_variance_flag": false},
    "pride": {"score": -0.78, "confidence": 0.63, "variance": 0.68, "high_variance_flag": false},
    "lust": {"score": -0.35, "confidence": 0.58, "variance": 0.52, "high_variance_flag": false},
    "wrath": {"score": -2.22, "confidence": 0.78, "variance": 1.56, "high_variance_flag": false},
    "gluttony": {"score": 0.25, "confidence": 0.53, "variance": 0.38, "high_variance_flag": false},
    "envy": {"score": -0.45, "confidence": 0.55, "variance": 0.18, "high_variance_flag": false},
    "sloth": {"score": 0.75, "confidence": 0.63, "variance": 0.56, "high_variance_flag": false}
  },
  "quality": {
    "score": 76.2,
    "tier": "moderate",
    "components": {
      "consistency": 78.5,
      "variance": 88.0,
      "style": 100.0,
      "engagement": 72.0
    }
  },
  "response_styles": {
    "ers": {"flag": false, "percentage": 8.3},
    "mrs": {"flag": false, "percentage": 21.4}
  },
  "flags": []
}
```

---

**F.2 User B: "The Direct One"**

**Questionnaire Responses:**

**Q1 (Group Dinner Check):**

> I always track what I ordered and pay exactly that. Why should I subsidize someone else's expensive steak when I had a salad? I'm not afraid to speak up about it either — it's about fairness. If people get uncomfortable, that's their problem. I've worked hard for my money.

**Q4 (Unequal Split):**

I'd call them out directly. Everyone deserves to know who's actually contributing and who's coasting. I've done this before and yes, it's uncomfortable, but someone has to say something. I usually end up leading these conversations because nobody else will step up.

**Q6 (Feedback Received):**

I'd listen, but I'd also push back if I disagreed. Not everything someone says is automatically true just because they're being "honest." I know myself pretty well. If there's validity to it, I'll consider it, but I'm not going to just accept criticism without evaluating it critically.

*(Q2, Q3, Q5 omitted for brevity — similar assertive pattern)*

---

**Final Profile for User B:**

json
```
{
  "user_id": "user_b",
  "version": 1,
  "created_at": "2026-01-29T15:00:00Z",
  "sins": {
    "greed": {"score": 2.15, "confidence": 0.82, "variance": 0.95, "high_variance_flag": false},
    "pride": {"score": 2.85, "confidence": 0.85, "variance": 0.72, "high_variance_flag": false},
    "lust": {"score": 0.42, "confidence": 0.55, "variance": 0.88, "high_variance_flag": false},
    "wrath": {"score": 2.65, "confidence": 0.88, "variance": 0.65, "high_variance_flag": false},
    "gluttony": {"score": 0.35, "confidence": 0.52, "variance": 0.45, "high_variance_flag": false},
    "envy": {"score": 1.25, "confidence": 0.68, "variance": 0.92, "high_variance_flag": false},
    "sloth": {"score": -1.55, "confidence": 0.75, "variance": 0.58, "high_variance_flag": false}
  },
  "quality": {
    "score": 81.5,
    "tier": "high"
  }
}
```

---

**F.3 Similarity Calculation: A ↔ B**

**Step 1: Check each trait for shared direction**

| Sin | Score A | Score B | A Direction | B Direction | Shared? |
|---|---|---|---|---|---|
| greed | -1.35 | +2.15 | virtue | vice | ❌ No |
| pride | -0.78 | +2.85 | virtue | vice | ❌ No |
| lust | -0.35 | +0.42 | neutral | neutral | ❌ No (both neutral) |
| wrath | -2.22 | +2.65 | virtue | vice | ❌ No |
| gluttony | +0.25 | +0.35 | neutral | neutral | ❌ No (both neutral) |
| envy | -0.45 | +1.25 | neutral | vice | ❌ No |
| sloth | +0.75 | -1.55 | vice | virtue | ❌ No |

**Step 2: Calculate contribution (0 overlaps)**

No traits share direction, so:

- Raw contribution = 0
- Raw score = 0 / 7.4 = 0

**Step 3: Apply quality multiplier**

Quality A = 76.2, Quality B = 81.5 Both ≥ 60, so multiplier = 1.0

**Step 4: Final result**

json
```
{
  "raw_score": 0.0,
  "adjusted_score": 0.0,
  "quality_multiplier": 1.0,
  "breakdown": [],
  "overlap_count": 0,
  "tier": "weak",
  "display_mode": "limited"
}
```

**Interpretation:** User A (The Peacemaker) and User B (The Direct One) have opposite personality profiles on nearly every dimension. They would likely experience significant friction in a relationship due to mismatched conflict styles, financial attitudes, and ego dynamics.

### F.4 High-Similarity Example: A ↔ C

**User C Profile (similar to A):**

json
```
{
  "user_id": "user_c",
  "sins": {
    "greed": {"score": -1.80, "confidence": 0.80},
    "pride": {"score": -0.95, "confidence": 0.70},
    "lust": {"score": 0.25, "confidence": 0.55},
    "wrath": {"score": -2.85, "confidence": 0.90},
    "gluttony": {"score": 0.55, "confidence": 0.58},
    "envy": {"score": -0.85, "confidence": 0.62},
    "sloth": {"score": 1.20, "confidence": 0.72}
  },
  "quality": {"score": 79.0, "tier": "moderate"}
}
```

**Overlap calculation:**

| Sin | Score A | Score C | Shared? | Weight | Contribution |
|-----|---------|---------|---------|--------|--------------|
| greed | -1.35 | -1.80 | ✅ virtue | 0.9 | 0.078 |
| pride | -0.78 | -0.95 | ✅ virtue | 1.2 | 0.098 |
| wrath | -2.22 | -2.85 | ✅ virtue | 1.5 | 0.182 |
| sloth | +0.75 | +1.20 | ✅ vice | 1.3 | 0.108 |

**Calculation for wrath:**
```
Trait similarity = 1 - (|-2.22 - (-2.85)| / 10) = 1 - (0.63 / 10) = 0.937
Avg confidence = (0.78 + 0.90) / 2 = 0.84

Contribution = 0.937 × 0.84 × 1.5 = 1.18 → normalised: 0.182
```

**Final result:**

json
```
{
  "raw_score": 0.466,
  "adjusted_score": 0.466,
  "breakdown": [
    {"sin": "greed", "direction": "virtue", "contribution": 0.078},
    {"sin": "pride", "direction": "virtue", "contribution": 0.098},
    {"sin": "wrath", "direction": "virtue", "contribution": 0.182},
    {"sin": "sloth", "direction": "vice", "contribution": 0.108}
  ],
  "overlap_count": 4,
  "tier": "moderate",
  "display_mode": "full"
}
```

**Interpretation:** User A and User C share 4 trait directions, with particularly strong alignment on conflict avoidance (wrath). This suggests good compatibility for daily interactions and conflict resolution.

---

## Appendix G: Sin Weight Rationale

The following weights were determined by the Harmonia team based on relationship psychology principles. These are informed hypotheses, not empirically validated coefficients. Future iterations should test and adjust weights based on matching outcome data.

| Sin | Weight | Team Rationale |
|---|---|---|
| **Wrath** | 1.5 | Conflict style is the strongest predictor of day-to-day relationship friction. Couples with mismatched approaches to disagreement (one avoidant, one confrontational) report consistent communication problems. This aligns with Gottman's research identifying criticism, contempt, and defensiveness as relationship-damaging patterns. |
| **Sloth** | 1.3 | Motivation and energy alignment affects lifestyle compatibility. Mismatched activity levels create friction in household responsibilities, social plans, and life goals. One partner wanting adventure while the other wants rest causes recurring tension. |
| **Pride** | 1.2 | Ego dynamics affect power balance and communication. Status-seeking or validation needs can create competition rather than partnership. However, some pride mismatch is tolerable (one partner can be more humble). |
| **Lust** | 1.0 | Novelty-seeking and spontaneity affect activity preferences and intimacy patterns. Used as baseline weight. Important but often negotiable — couples can find middle ground on adventure vs. routine. |

| | | |
|---|---|---|
| **Greed** | 0.9 | Financial attitudes matter but are often explicitly negotiated. Couples can establish shared budgets and financial plans even with different natural tendencies. Open communication mitigates mismatch. |
| **Gluttony** | 0.8 | Moderation preferences affect lifestyle but are lower-stakes than conflict or motivation. Differences in indulgence (food, spending, leisure) are noticeable but rarely relationship-breaking. |
| **Envy** | 0.7 | Comparison tendencies are least predictive of relationship outcomes among these traits. Can manifest positively (shared ambition) or negatively (resentment). Less direct impact on partner interactions. |

**Important Caveats:**

1. These weights have not been validated against relationship outcome data
2. The Seven Deadly Sins framework itself is novel and unvalidated in academic literature
3. Individual weight values should be treated as adjustable parameters
4. Post-launch data collection should inform weight recalibration
5. Cultural context may affect which traits matter most

**Future Validation Plan:**

1. Collect match feedback (did you message? did you meet? satisfaction rating)
2. Correlate trait overlap by sin with positive outcomes
3. Adjust weights based on empirical contribution to match success
4. Consider per-user weight learning (some users may care more about certain traits)

---

## Appendix H: Glossary

| Term | Definition |
|---|---|
| **Aggregation** | The process of combining 6 question-level scores into a single trait score using confidence-weighted mean. |
| **Confidence** | A 0.0-1.0 value indicating Gemini's certainty in a parsed score. Higher confidence means clearer linguistic signals in the response. |
| **CWMV** | Confidence-Weighted Mean with Variance. Aggregation method that weights item scores by their confidence values, used when confidence varies meaningfully across items. |
| **Discrepancy** | A flag raised when a user's claimed trait (e.g., "I never get angry") contradicts linguistic signals in their response (e.g., using anger words). |

| | |
|---|---|
| **Display Mode** | How much similarity detail to show users: "full" (all overlaps), "limited" (summary only), "hidden" (no personality data shown). |
| **ERS** | Extreme Response Style. A response pattern where >40% of scores are at extremes (±4 or ±5), suggesting possible careless responding or genuinely extreme personality. |
| **Evidence** | A direct quote from the user's response that supports the Gemini-assigned score for a trait. |
| **High Variance Flag** | Indicator that a trait's scores varied significantly (SD > 3.0) across the 6 questions, reducing confidence in the aggregated score. |
| **LIWC Signals** | Linguistic Inquiry and Word Count features extracted from responses: pronoun usage, emotion words, certainty markers. Used as secondary validation. |
| **MRS** | Midpoint Response Style. A response pattern where >50% of scores are neutral (0), especially combined with fast completion time, suggesting disengagement. |
| **Neutral Threshold** | The score range (±0.5) within which trait scores don't indicate a clear direction. Neutral scores don't contribute to similarity overlap. |
| **Overlap** | When two users share the same direction (both virtue OR both vice) on a trait with scores outside the neutral threshold. |
| **PIIP** | Personality Inference through Interactive Prompting. Felix's framework for the questionnaire design, using scenario-based questions to elicit natural responses. |
| **Positive Overlap** | The similarity calculation principle where only shared trait directions contribute to the score. Differences are ignored, not penalised. |
| **Profile** | The aggregated personality data for a user, containing 7 sin scores, quality metrics, and metadata. |
| **Quality Score** | A 0-100 composite score indicating profile reliability, based on consistency, variance, response style, and engagement. |
| **Quality Tier** | Classification of profile reliability: "high" (≥80), "moderate" (60-79), "low" (<60), or "rejected" (<4 valid questions). |
| **Sin** | One of the seven personality dimensions in the framework: greed, pride, lust, wrath, gluttony, envy, sloth. |

| | |
|---|---|
| **Stage 2** | The personality matching stage in Harmonia's cascaded pipeline, between visual (Stage 1) and HLA genetic (Stage 3). |
| **Stage 2 Threshold** | The minimum similarity score (default 0.40) required for a match to proceed to Stage 3. |
| **Trait** | Synonym for "sin" in the context of the Seven Deadly Sins personality framework. |
| **Vice** | The positive end of a sin scale (scores > +0.5), indicating the trait is expressed in its traditional "sin" form. |
| **Virtue** | The negative end of a sin scale (scores < -0.5), indicating the opposite of the sin (e.g., generosity for greed, humility for pride). |
| **Weight** | The relative importance of each sin in similarity calculations. Wrath (1.5) contributes most; envy (0.7) contributes least. |

---

## Appendix I: External References

### Academic Research

**Foundational:**

- Joel, S., Eastwick, P. W., & Finkel, E. J. (2017). Is Romantic Desire Predictable? Machine Learning Applied to Initial Romantic Attraction. *Psychological Science*, 28(10), 1478-1489.
    - *Key finding: Pre-meeting profile data cannot predict relationship-specific chemistry*
- Gottman, J. M., & Silver, N. (1999). *The Seven Principles for Making Marriage Work*. Crown Publishers.
    - *Framework for conflict patterns in relationships*

**Personality & Relationships:**

- Back, M. D., et al. (2011). NARCISSUS REVEALED: Narcissistic characteristics and relationship outcomes. *Journal of Personality and Social Psychology*.
- Montoya, R. M., Horton, R. S., & Kirchner, J. (2008). Is actual similarity necessary for attraction? A meta-analysis of actual and perceived similarity. *Journal of Social and Personal Relationships*.

**LLM Personality Inference:**

- No peer-reviewed validation exists for LLM-based personality inference from dating scenario responses as of January 2026. Harmonia's approach should be considered experimental.

**Technical Documentation**

**Gemini API:**

- Google AI for Developers: https://ai.google.dev/docs
- Gemini API Reference: https://ai.google.dev/api
- Structured Output Guide: https://ai.google.dev/gemini-api/docs/structured-output

**FastAPI:**

- Official Documentation: https://fastapi.tiangolo.com/
- Async Testing: https://fastapi.tiangolo.com/advanced/async-tests/
- Dependency Injection: https://fastapi.tiangolo.com/tutorial/dependencies/

**Railway:**

- Documentation: https://docs.railway.app/
- Health Checks: https://docs.railway.com/reference/healthchecks
- Private Networking: https://docs.railway.com/reference/private-networking
- Pricing: https://docs.railway.com/reference/pricing/plans

**Testing:**

- pytest: https://docs.pytest.org/
- pytest-asyncio: https://pytest-asyncio.readthedocs.io/
- httpx (async HTTP client): https://www.python-httpx.org/

**Python Libraries**

| Library | Version | Purpose |
|---|---|---|
| google-generativeai | ≥0.8.0 | Gemini API SDK |
| fastapi | ≥0.115.0 | Web framework |
| uvicorn | ≥0.30.0 | ASGI server |
| pydantic | ≥2.9.0 | Data validation |
| structlog | ≥24.4.0 | Structured logging |
| tenacity | ≥9.0.0 | Retry logic |

| | | |
|---|---|---|
| jsonrepair | ≥0.1.0 | JSON fixing |
| pytest | ≥8.3.0 | Testing framework |
| pytest-asyncio | ≥0.24.0 | Async test support |
| httpx | ≥0.27.0 | Async HTTP client |

---

**End of Appendices**