1. V2 streaming analysis — high-level overview

2. This plan turns your draft into an executable roadmap: clear contracts, prioritized service migration, test gates, rollout strategy, and concrete implementation tasks for each milestone. It's organized around incremental delivery (fast visible wins first), strong contract tests (prevent UI breakage), and operational readiness (metrics, fallbacks, privacy).

3. Architecture and contracts (what must be true)

4. Single source-of-truth registry: /v2/services returns [{name, version, supports_streaming}] for frontend feature gating.
5. AnalysisService protocol: all services must implement analyze(...) and optionally stream_analyze(...). Each result must include: service_name, service_version, local, gemini (or null), errors (or null), timings.
6. SSE event envelope (stable): event, session_id, sequence_id, service, type (partial|final|meta), version=v2, payload, meta. sequence_id monotonic per session.
7. GeminiClientV2 abstraction: centralizes model selection, streaming vs batch, retries, JSON-only requests + robust parsing/repair, metrics hooks, and a shared async HTTP client.
8. StreamingCoordinator: translates SDK streaming into normalized transcription and analysis chunks with deterministic chunking rules.
9. Privacy boundary: no raw text/audio in logs; only IDs, sizes, and hashes. Sanitizer + structured logging mandatory.

10. Roadmap — phases, sprint-sized milestones

11. Goal: ship first useful v2 experience in 4 sprints (2-week cadence recommended). Each sprint ends with tests and a deployable feature flag.
12. Sprint 0 — Prep (1 week)
13. Create analysis_protocol.py and registry shapefile.
14. Add /v2/services endpoint (feature-map only).
15. Implement log_sanitizer and logging_utils (basic stubs).
16. CI scaffolding for contract tests.
17. Sprint 1 — Core infra + Quantitative metrics (2 weeks)
18. Implement GeminiClientV2 skeleton (no external SDK calls; plumb metrics & retries).
19. Implement StreamingCoordinator.transcription stub + transcript chunker.
20. Implement QuantitativeMetricsService full (streaming partials + final) and unit tests.
21. Runner.stream_run wiring for transcription + audio_quality early event; SSE contract tests.
22. Frontend: consume /v2/services, show quantitative_metrics panel with partials.
23. Sprint 2 — Linguistic + Audio Analysis + coordinator improvements (2 weeks)
24. Implement LinguisticService (batch).
25. Implement AudioAnalysisService (local audio features; no Gemini multi-modal yet).
26. Enhance StreamingCoordinator to support fallback (emit meta if no streaming).
27. Add contract tests (service shape + sequence ordering).
28. End-to-end integration test: audio -> transcription -> services -> SSE.
29. Sprint 3 — Manipulation + Gemini JSON path (2 weeks)
30. Complete GeminiClientV2.query_json with JSON parse & repair logic and model fallback.
31. Implement ManipulationService using gemini injection; stream_analyze emits partial locals.
32. Stress tests: runner concurrency under simulated high-load; sequence_id checks.
33. Add metrics dashboard (calls, latencies, fallbacks, errors).
34. Sprint 4 — EnhancedUnderstanding + polish + rollout (2 weeks)
35. Implement EnhancedUnderstandingService (LLM-backed summaries & follow-ups).
36. Finalize API docs, contract tests, and migration plan.
37. Feature-flagged rollout: route /v2/analyze and /v2/analyze/stream behind flag; keep v1 archived.
38. Optional follow-ups (after initial rollout)
39. Gemini multi-modal analyze_audio improvements.
40. Per-service configurable model hints and quota tracking.
41. Streaming backpressure and batching tunables.

42. Implementation details and deliverables (developer-ready)

43. Registry
    - 43.1. File: backend/services/v2_services/registry.py
    - 43.2. Contract: REGISTRY: Dict[name, Type[AnalysisService]]; add metadata endpoint: /v2/services.
44. Analysis runner
    - 44.1. Expose run(...) (batch) and stream_run(...) (SSE).
    - 44.2. stream_run responsibilities: early audio_quality, stream transcription partials/final, then concurrent services (streaming-aware).
    - 44.3. Implement sequence_id as an atomic counter (async-safe) per session.
45. GeminiClientV2 (deliverables)
    - 45.1. Methods: transcribe(audio_bytes, model_hint=None) -> str; query_json(prompt, model_hint=None, max_tokens=...) -> Dict; analyze_audio(...)
    - 45.2. Prompt should be injectable object for getting analysis maybe?!
    - 45.3. Features: model_hint mapping; automatic retries with exponential backoff; JSON content-type enforcement; robust repair (try: json.loads; except: run a lightweight grammar repair pass + reparse); metrics hooks (prometheus counters or similar).
    - 45.4. Injected HTTP client (httpx.AsyncClient) shared across app.
46. StreamingCoordinator
    - 46.1. stream_transcription(audio) yields {"type":"transcription.partial|final","text", "sequence_id"} using deterministic chunking (e.g., sentence/token boundaries).
    - 46.2. stream_analysis(prompt_spec, context) yields normalized {"type":"analysis.partial|final","json"}; when real streaming not available, emits a meta interim then final.

47. Service pattern
    - 47.1. Services receive gemini instance via constructor injection for testability.
    - 47.2. Streaming services must adhere to partial-first semantics: emit meaningful local partials (small payloads) often, reserve heavy LLM JSON for finals.
    - 47.3. Deterministic local metrics must be computed identical in analyze(...) and stream_analyze(...) end-states.
48. SSE and frontend contract
    - 48.1. Minimal payloads: avoid resending whole transcript; only chunked text until final transcript event.
    - 48.2. Merge logic: frontend merges results by {service, sequence_id}. UI must maintain per-service loading states until final.
    - 48.3. Errors: analysis.error events with service-specific error payload without killing the session.
49. Privacy & logging
    - 49.1. log_sanitizer.sanitize_meta(meta) must be applied to all stored logs and emitted metrics.
    - 49.2. Structured logs: timestamp, session_id, sequence_id, service, event, level, sanitized_meta.
50. Testing
    - 50.1. Contract tests assert envelope shape and required keys.
    - 50.2. Determinism tests for local metrics using fixed fixtures.
    - 50.3. Runner tests simulate intermittent Gemini streaming failures and expect fallbacks (meta event + final).
    - 50.4. Load tests to validate concurrency and sequence ordering under parallel sessions.

51. Testing, rollout, and migration strategy

52. Feature-flagged endpoints: /v2/* behind toggle; deploy v2 inactive by default.
53. Side-by-side: keep v1 in archived folder until parity tests pass.
54. Contract-first rollout:
    - 54.1. Unit tests for protocol and services.
    - 54.2. Integration tests (runner + gemini stub) validating SSE order, sequence_id monotonicity, partial vs final.
    - 54.3. Staging rollout with small percentage of traffic; synthetic load and real-user sampling.
    - 54.4. Gradual ramp to 100% once error budgets and latency SLAs are satisfied.
55. Migration acceptance gates
    - 55.1. All /v2 services pass: protocol compliance tests, SSE schema tests, end-to-end integration tests.
    - 55.2. Latency: median 95th percentile service time within acceptable range vs v1.
    - 55.3. Error budget: <= agreed anomalies during staged rollout.
    - 55.4. Frontend compatibility: verify UI rendering for partials, finals, and errors.

56. Risks, mitigations, and operational notes

57. Risk: out-of-order events across concurrent service streams.
    - 57.1. Mitigation: sequence_id is global and monotonic; runner must increment atomically before emitting any event.
58. Risk: large LLM JSON payloads slow UI.
    - 58.1. Mitigation: keep final gemini JSON small; enforce schema size limits; compress/trim optional verbose fields before SSE.
59. Risk: Gemini streaming unavailable / SDK breakage.
    - 59.1. Mitigation: StreamingCoordinator fallback emits meta interim + final; implement graceful model fallback and retry; metric for fallback rate.
60. Risk: privacy leakage via logs.
    - 60.1. Mitigation: strict sanitizer, e2e test to assert no raw_text fields make it into logs; review at code PRs.

62. Risk: resource exhaustion under load (many concurrent LLM calls).
    62.1. Mitigation: concurrency limits per-run, global rate limiter for GeminiClientV2, circuit breaker on repeated LLM failures.

63. Immediate next steps (actionable)

64. Create the analysis_protocol.py and add type stubs + contract tests (Sprint 0 priority).
65. Implement /v2/services metadata endpoint and registry skeleton.
66. Wire a minimal GeminiClientV2 skeleton (no external calls) with metrics hooks and unit tests.
67. Implement QuantitativeMetricsService fully and integrate with stream_run path so frontend sees live partials (Sprint 1 objective).
68. Add CI checks: envelope shape tests, sequence monotonicity assertions, sanitizer checks.
69.

V2 streaming protocol and architecture draft

70. Core contracts and event model

71. Analysis service protocol

72. Each analysis module implements a single protocol so the runner can orchestrate them generically and concurrently. The protocol is idempotent, testable, and streaming-aware.
73. # backend/services/v2_services/analysis_protocol.py
from typing import AsyncGenerator, Dict, Any, Optional, Protocol

class AnalysisService(Protocol):
  service_name: str
  service_version: str
  supports_streaming: bool

  async def analyze(
    self,
    transcript: str,
    audio: Optional[bytes],
    meta: Dict[str, Any],
) -> Dict[str, Any]:
    """
    Returns:
      {
       "service_name": "...",
       "service_version": "...",
       "local": {...},   # deterministic metrics
       "gemini": {...} | None, # parsed JSON from Gemini
       "errors": None | {...}, # structured errors
       "timings": {...}   # ms counters
      }
    """

  async def stream_analyze(
    self,
    transcript_stream: AsyncGenerator[Dict[str, Any], None],
    audio: Optional[bytes],
    meta: Dict[str, Any],
) -> AsyncGenerator[Dict[str, Any], None]:
    """
    Yields partial results with the same envelope shape as `analyze`, plus:
      {"partial": True, "sequence_id": int}
    Ends with a final:
      {"partial": False, "sequence_id": int, ...}
    """

74. Streaming event envelope for SSE

75. Runner emits normalized events that the frontend can merge deterministically. Event names stay consistent; payloads stay small.
76. {
 "event": "analysis.update",   // or analysis.error, analysis.done
 "session_id": "uuid",
 "sequence_id": 42,
 "service": "quantitative_metrics", // or "transcription", "audio_quality", etc.
 "type": "partial",   // partial | final | meta
 "version": "v2",
 "payload": { /* service result fragment */ },
 "meta": { "elapsed_ms": 1200 }
}
77. Ordering: sequence_id ensures stable rendering.
78. Boundaries: type makes “partial vs final” explicit.
79. Isolation: Errors surface as analysis.error without crashing the run.

80. Gemini V2 client and streaming layer

81. Gemini client (centralized)

82. All LLM calls flow through a single module for model selection, retries, parsing, and metrics. Services never touch SDKs directly.
83. # backend/services/v2_services/gemini_client.py
from typing import Dict, Any, Optional

class GeminiClientV2:
  def __init__(self, config, http_client=None):
    self.config = config
    self.http = http_client or ... # shared async httpx client
    # metrics: counters for calls, latency, fallbacks

  async def transcribe(self, audio_bytes: bytes, *, model_hint: Optional[str]=None) -> str:
    # choose model; stream if available; return final transcript
    ...

  async def query_json(self, prompt: str, *, model_hint: Optional[str]=None, max_tokens: int=2048) -> Dict[str, Any]:
    # request application/json; robustly parse/repair JSON; fallback models and retries

```
        ...

    async def analyze_audio(
        self,
        audio_bytes: bytes,
        transcript: str,
        prompt: str,
        *,
        model_hint: Optional[str]=None,
        max_tokens: int=4096
    ) -> Dict[str, Any]:
        # multi-modal when available; JSON-normalized
output
        ...
```

84. Streaming coordinator (normalizing raw chunks)

85. A dedicated layer translates SDK streaming into normalized transcript and analysis events for services.

86.
```
# backend/services/v2_services/streaming_coordinator.py
from typing import AsyncGenerator, Dict, Any, Optional

class GeminiStreamingCoordinator:
    def __init__(self, gemini: GeminiClientV2, config:
Dict[str, Any]):
        self.gemini = gemini
        self.config = config

    async def stream_transcription(self, audio_bytes: bytes)
-> AsyncGenerator[Dict[str, Any], None]:
        # yields {"type": "transcription.partial", "text": "...",
"sequence_id": n}
        # ends with {"type": "transcription.final", "text": "...",
"sequence_id": m}
        ...

    async def stream_analysis(
        self,
        prompt_spec: Dict[str, Any],
        context: Dict[str, Any],
    ) -> AsyncGenerator[Dict[str, Any], None]:
        # yields normalized {"type": "analysis.partial|final",
"json": {...}}
        ...
```

87. PromptSpec: Services declare their prompt and output schema as data, not code; the coordinator turns that into Gemini requests.

88. Fallbacks: When streaming isn't available, coordinator emits one small interim "meta" event and then a final.

89. Runner, registry, and compliance

90. Service registry and capabilities map

91. A single source of truth that both backend and frontend can reference for available services, versions, and streaming support.

92.
```
# backend/services/v2_services/registry.py
from typing import Dict, Type
from .analysis_protocol import AnalysisService
from .impl import (
    ArgumentService,
    AudioAnalysisService,
    AudioService,
    ConversationFlowService,
    EnhancedUnderstandingService,
    LinguisticService,
    ManipulationService,
    PsychologicalService,
    QuantitativeMetricsService,
    SessionInsightsService,
    SpeakerAttitudeService,
)

REGISTRY: Dict[str, Type[AnalysisService]] = {
    "argument": ArgumentService,
    "audio_analysis": AudioAnalysisService,
    "audio": AudioService,
    "conversation_flow": ConversationFlowService,
    "enhanced_understanding":
EnhancedUnderstandingService,
    "linguistic": LinguisticService,
    "manipulation": ManipulationService,
    "psychological": PsychologicalService,
    "quantitative_metrics": QuantitativeMetricsService,
    "session_insights": SessionInsightsService,
    "speaker_attitude": SpeakerAttitudeService,
}
```

93. Expose a thin API route that returns { service_name, version, supports_streaming }[] so the frontend knows what to expect.

94. Analysis runner with streaming

95. The runner orchestrates transcription, service execution (concurrent), merging, and SSE emission. It is SDK-agnostic and error-resilient.

96.
```
# backend/services/v2_services/analysis_runner.py
import asyncio
from typing import Dict, Any, Optional, AsyncGenerator,
List
from .registry import REGISTRY
from .gemini_client import GeminiClientV2
from .streaming_coordinator import
GeminiStreamingCoordinator

class AnalysisRunner:
    def __init__(self, config):
        self.config = config
        self.gemini = GeminiClientV2(config)
        self.streamer =
GeminiStreamingCoordinator(self.gemini, config)

    async def run(
        self,
        audio_bytes: bytes,
        transcript: Optional[str],
        meta: Dict[str, Any],
        service_names: Optional[List[str]] = None,
    ) -> Dict[str, Any]:
        # batch run (no SSE)
        final: Dict[str, Any] = {
            "session_id": meta.get("session_id"),
            "transcript": transcript or await
self.gemini.transcribe(audio_bytes),
            "audio_quality": await
self._compute_audio_quality(audio_bytes),
```

```
        "services": {},
        "timings": {},
        "errors": {},
    }

    async def _call(service_cls):
        svc = service_cls()
        try:
            result = await svc.analyze(final["transcript"],
audio_bytes, meta)
            final["services"][svc.service_name] = result
        except Exception as e:
            final["errors"][svc.__name__] = {"message": str(e)}
    tasks = [asyncio.create_task(_call(REGISTRY[name]))
for name in (service_names or REGISTRY.keys())]
    await asyncio.gather(*tasks, return_exceptions=True)
    return final

  async def stream_run(
    self,
    audio_bytes: bytes,
    meta: Dict[str, Any],
    service_names: Optional[List[str]] = None,
  ) -> AsyncGenerator[Dict[str, Any], None]:
    sequence_id = 0

    # 1) audio quality early
    aq = await self._compute_audio_quality(audio_bytes)
    yield self._event("analysis.update",
sequence_id:=sequence_id+1, "audio_quality", "final",
{"audio_quality": aq}, meta)

    # 2) stream transcription to UI, while buffering
transcript for services
    transcript_buffer = []
    async for t_evt in
self.streamer.stream_transcription(audio_bytes):
        payload = {"transcript_chunk": t_evt.get("text", "")}
        yield self._event("analysis.update",
sequence_id:=sequence_id+1, "transcription", "partial",
payload, meta)
        if t_evt["type"].endswith(".final"):
            full_transcript = t_evt["text"]
            yield self._event("analysis.update",
sequence_id:=sequence_id+1, "transcription", "final",
{"transcript": full_transcript}, meta)
            break

    # 3) launch services concurrently
    names = service_names or list(REGISTRY.keys())
    async def _stream_service(name: str):
        svc = REGISTRY[name]()
        try:
            if getattr(svc, "supports_streaming", False):
                async for part in
svc.stream_analyze(self._transcript_stream(full_transcrip
t), audio_bytes, meta):
                    part["partial"] = True
                    yield self._event("analysis.update",
sequence_id:=sequence_id+1, name, "partial", part, meta)
                final = await svc.analyze(full_transcript,
audio_bytes, meta)
                yield self._event("analysis.update",
sequence_id:=sequence_id+1, name, "final", final, meta)
            else:
                final = await svc.analyze(full_transcript,
audio_bytes, meta)
                yield self._event("analysis.update",
sequence_id:=sequence_id+1, name, "final", final, meta)
        except Exception as e:
            yield self._event("analysis.error",
sequence_id:=sequence_id+1, name, "final", {"error":
str(e)}, meta)

    awaitables =
[self._stream_to_sse(_stream_service(n)) for n in names]
    await asyncio.gather(*awaitables)

    yield {"event": "analysis.done", "session_id":
meta.get("session_id"), "sequence_id": sequence_id,
"version": "v2"}

  async def _compute_audio_quality(self, audio_bytes:
bytes) -> Dict[str, Any]:
    # duration/sample_rate/score computed locally
without LLM
    ...

  def _event(self, event, sequence_id, service, type_,
payload, meta):
    return {
      "event": event,
      "session_id": meta.get("session_id"),
      "sequence_id": sequence_id,
      "service": service,
      "type": type_,
      "version": "v2",
      "payload": payload,
      "meta": {"received_at": meta.get("received_at")},
    }

  async def _stream_to_sse(self, agen):
    async for evt in agen:
      yield evt

  async def _transcript_stream(self, text: str):
    # split by sentences/turns to feed streaming services
incrementally
    for i, chunk in enumerate(self._chunk_text(text)):
      yield {"text": chunk, "index": i}
```

97. Service implementations and responsibilities

98. Below are first-draft skeletons with clear domain boundaries. Each service returns deterministic local metrics plus optional Gemini JSON.

99. Quantitative metrics service

100.
```
# backend/services/v2_services/impl/quantitative_metrics_service.py
from typing import Dict, Any, Optional
from ..analysis_protocol import AnalysisService
```

```
class QuantitativeMetricsService(AnalysisService):
  service_name = "quantitative_metrics"
  service_version = "2.0.0"
  supports_streaming = True

  async def analyze(self, transcript: str, audio:
Optional[bytes], meta: Dict[str, Any]) -> Dict[str, Any]:
    local = self._compute_local(transcript)
    return {"service_name": self.service_name,
"service_version": self.service_version, "local": local,
"gemini": None, "errors": None, "timings": {}}

  async def stream_analyze(self, transcript_stream,
audio, meta):
    # incrementally update counts (word_count,
hesitation_count, etc.)
    agg = {"word_count": 0, "hesitation_count": 0,
"sentence_count": 0}
    async for chunk in transcript_stream:
      partial = self._update_agg(agg, chunk["text"])
      yield {"service_name": self.service_name,
"service_version": self.service_version, "local": partial,
"gemini": None, "errors": None, "timings": {}}

  def _compute_local(self, text: str) -> Dict[str, Any]:
    # robust tokenization, sentence splitting, filler
detection
    ...
  def _update_agg(self, agg, text):
    ...
```

101. Linguistic service

```
# backend/services/v2_services/impl/linguistic_service.py
class LinguisticService(AnalysisService):
  service_name = "linguistic"
  service_version = "2.0.0"
  supports_streaming = False

  async def analyze(self, transcript, audio, meta):
    local = {
      "formality_score": self._formality(transcript),
      "complexity_score": self._complexity(transcript),
      "repetition_count": self._repetitions(transcript),
    }
    return {"service_name": self.service_name,
"service_version": self.service_version, "local": local,
"gemini": None, "errors": None, "timings": {}}
```

103. Manipulation service (with Gemini JSON)

```
#
backend/services/v2_services/impl/manipulation_service
.py
class ManipulationService(AnalysisService):
  service_name = "manipulation"
  service_version = "2.0.0"
  supports_streaming = True

  def __init__(self, gemini=None):
    self.gemini = gemini  # injected by runner or factory

  async def analyze(self, transcript, audio, meta):
    local = self._local_indicators(transcript)
    prompt_spec = self._prompt_spec(transcript, meta)
    gemini_json = await
self.gemini.query_json(prompt_spec["prompt"],
model_hint=prompt_spec["model_hint"])
    return {"service_name": self.service_name,
"service_version": self.service_version, "local": local,
"gemini": gemini_json, "errors": None, "timings": {}}

  async def stream_analyze(self, transcript_stream,
audio, meta):
    # optional: emit partial local indicators before final
LLM JSON
    agg = {"manipulation_score": 0, "tactics": []}
    async for chunk in transcript_stream:
      yield {"service_name": self.service_name,
"service_version": self.service_version, "local":
self._update(agg, chunk["text"]), "gemini": None, "errors":
None, "timings": {}}

  def _prompt_spec(self, transcript, meta):
    return {"prompt": f"...JSON-only analysis of
manipulation tactics...\nTEXT:\n{transcript}",
"model_hint": "gemini-2.5-pro"}
```

105. Repeat this pattern for:
106. ArgumentService: argument strengths/weaknesses, coherence scoring.
107. AudioAnalysisService: pitch/pace/pauses from audio features locally, optional Gemini multi-modal summary.
108. AudioService: transcription (if made a service) and audio preprocessing summaries.
109. ConversationFlowService: turn-taking, interruptions, topic shifts.
110. EnhancedUnderstandingService: inconsistencies, evasiveness, suggested follow-ups.
111. PsychologicalService: emotions, motivation, stress indicators (LLM).
112. SessionInsightsService: deltas across time windows and sessions.
113. SpeakerAttitudeService: respect/sarcasm detectors and tone indicators.

114. Logging, sanitization, and privacy

115. Log sanitizer: Replace raw text/audio with hashes, sizes, and redaction marks.
116.
```
# backend/services/v2_services/log_sanitizer.py
def sanitize_meta(meta: dict) -> dict:
  # keep session_id, lengths, model ids; strip raw content
  ...
```
117. Logging utils: Structured logs per event and per service with levels.
118.
```
# backend/services/v2_services/logging_utils.py
def log_event(event: dict, level: str = "info"):
  # write structured JSON logs with sequence_id, service,
elapsed_ms
  ...
```
119. Never log raw transcripts or audio. Only sizes and IDs.

120. API, transport, and frontend contract

121. Backend routes

122. /v2/analyze: batch endpoint; returns final aggregated JSON.
123. /v2/analyze/stream: SSE; emits analysis.update|error|done with event envelope above.
124. /v2/services: list available services { name, version, supports_streaming }.

125. Frontend expectations

126. Merge logic: By service and sequence_id. Maintain per-panel loading until a final arrives.
127. Small events: Avoid sending entire transcript repeatedly; stream chunks and one final transcript.
128. Error states: Render service-specific errors without collapsing the session.

129. Testing, quality gates, and migration

130. Tests

131. Protocol compliance: Ensure each service exposes required keys and shapes.
132. Gemini client: Retry/fallback ordering, JSON parsing robustness, metrics hooks.
133. Runner streaming: Sequence ordering, partial vs final semantics, error isolation.
134. Audio/linguistic local: Deterministic outputs across fixtures.
135. SSE schema: Contract tests so UI consumption is stable.

136. Migration strategy

137. Side-by-side: Keep v1 in backend/services/archived/ until v2 feature parity passes unit and integration tests.
138. Feature flag or new endpoints: /v2/* keeps blast radius low while the frontend adapts.
139. Documentation update: Mark v1 archived once v2 services produce equal or better outputs with green tests.

140. Implementation checklist (first pass)

141. Analysis protocol: Create analysis_protocol.py and enforce returns.
142. GeminiClientV2: Centralize model hints, fallback, JSON parsing, metrics.
143. StreamingCoordinator: Normalize transcript and analysis streaming events.
144. Registry: Single authoritative map, expose /v2/services.
145. Runner: Batch and streaming paths, concurrency, error isolation, SSE emission.
146. Services (phase 1):
    - 146.1. QuantitativeMetricsService: streaming partials + final metrics.
    - 146.2. LinguisticService: batch local analysis.
    - 146.3. ManipulationService: local + Gemini JSON.
    - 146.4. AudioAnalysisService: local audio features + optional Gemini summary.
147. Logging & Sanitization: No raw content, structured logs.
148. Tests: Protocol, client, runner streaming, services unit.
149. If you want me to start with code stubs, say Go and list the top 5 services to migrate immediately. I recommend: quantitative_metrics, linguistic, audio_analysis, manipulation, enhanced_understanding — they unlock the most visible UI improvements quickly.