Aspectarian — Software Specification

Version: 1.4 Draft
Author: ASTROinnovations
Framework: Jyotiṣa + Guṇa + Consciousness Mapping
Engine: Swiss Ephemeris (pyswisseph), Lahiri Ayanāṃśa
Intelligence: Anthropic Claude API (claude-sonnet-4)

---

1. Purpose

Aspectarian is a Vedic astrology transit interpretation system. Given a user's birth chart and the current positions of the grahas, it calculates and interprets the effects of active transits for a chosen window — day, week, or month — using classical Jyotiṣa rules, then delivers those interpretations through a conversational interface powered by a frontier language model.

The user receives a structured narrative report, then can ask follow-up questions in plain language. Claude holds the full astrological context for the session — natal chart, active contacts, and a summary of reading history — and responds as a Jyotiṣa interpreter grounded in the guṇa–consciousness framework encoded in `symbol_map.json`.

---

2. Scope

In Scope

- Natal chart computation (sidereal, Lahiri ayanāṃśa, whole-sign houses)
- Real-time and date-range transit computation
- Transit-to-natal contact analysis (Parāśaran dṛṣṭi + yuti) with strength scoring
- Retrograde detection and tone modification
- Moon's daily nakṣatra transit (day reports)
- System prompt assembly from chart data + `symbol_map.json` + reading history
- LLM-generated transit report (day / week / month)
- Conversational Q&A within a session

- Persistent user profile and natal chart (SQLite, v1)
- Reading history — lightweight summary in context (v1), full retrieval on demand (v2)
- South Indian square chart viewer (natal + transit overlay toggle)
- IAST transliteration throughout all output
- JSON (machine) + narrative text (human) output formats
- CLI interface (v1), web UI (v1), REST API + OAuth (v2)

User Model

- **v1:** Single-user, no authentication. One profile per installation. User manages their own chart.
- **v2:** Multi-user, OAuth (Google). Practitioner model: a Jyotiṣī can manage multiple client profiles and generate readings for any of them. Includes daśā awareness and daśā report window.

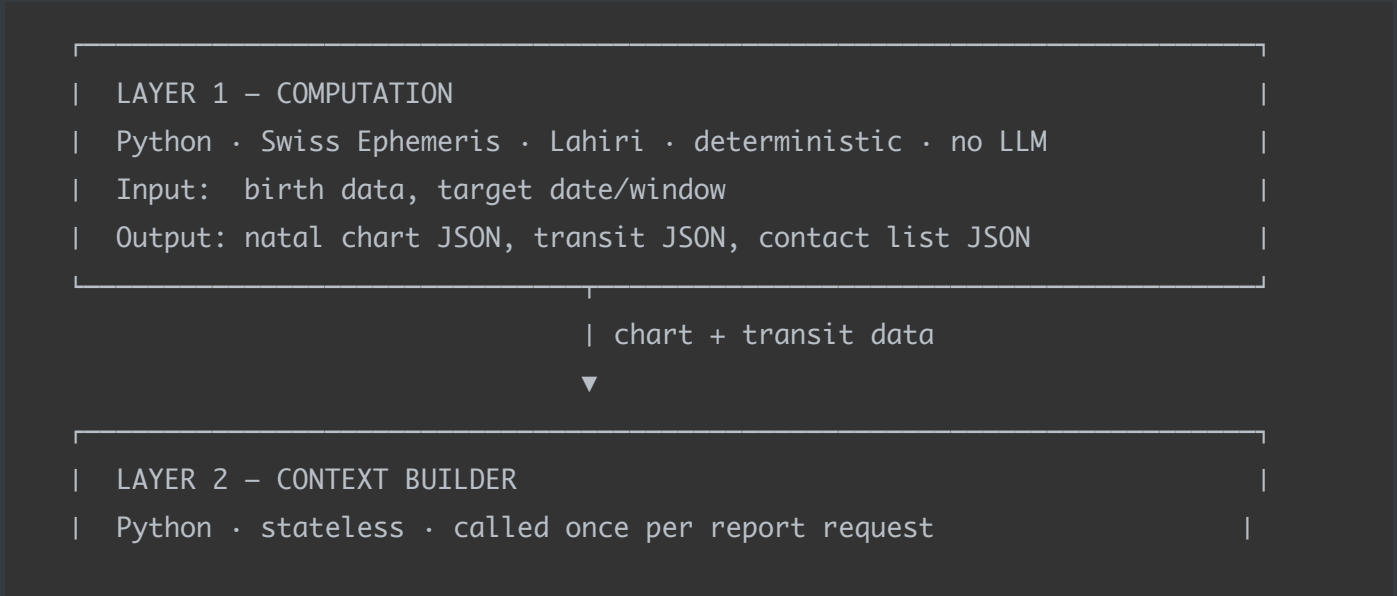
Out of Scope (v1)

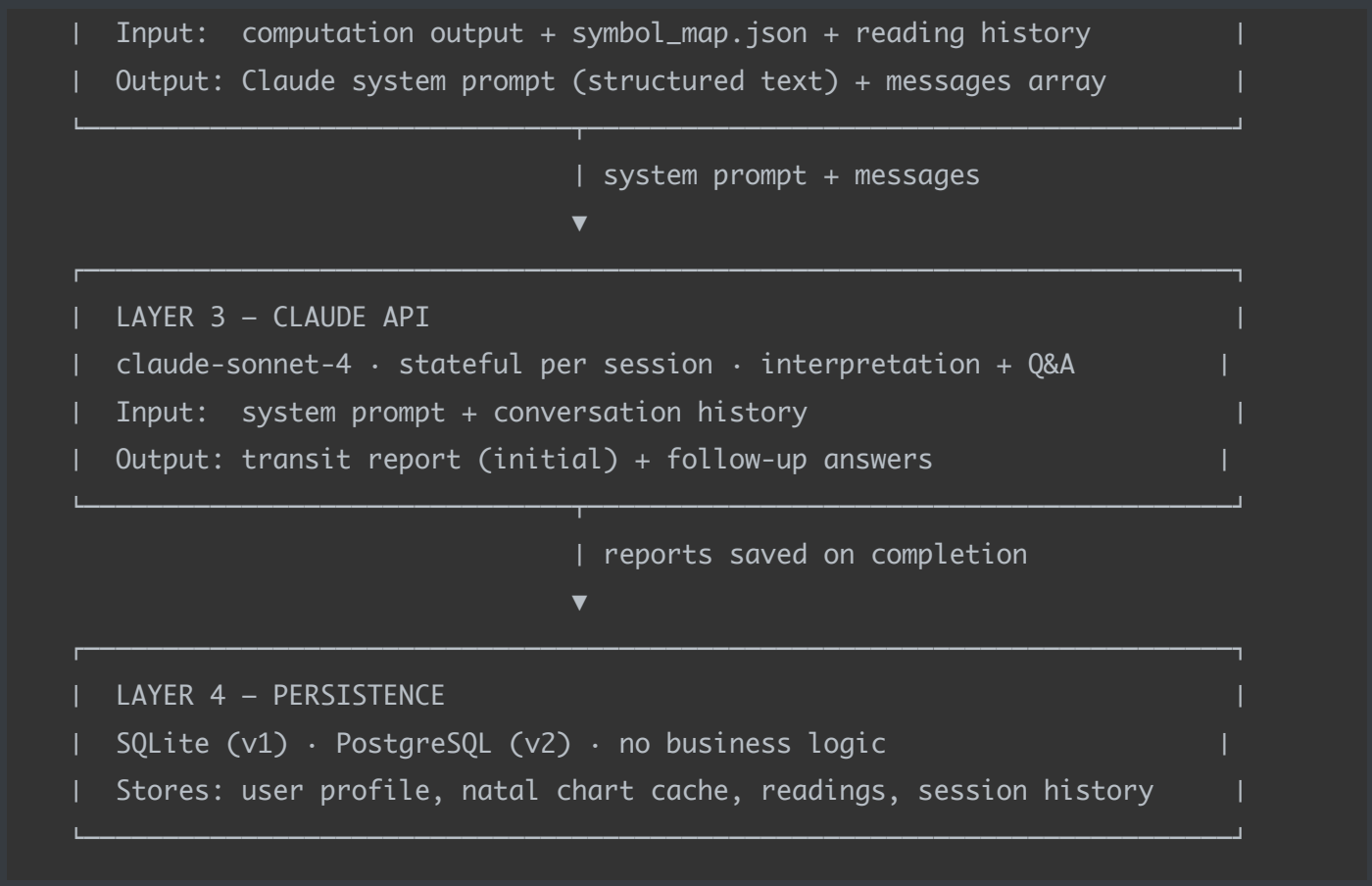
- Daśā / antaradaśā systems
- Divisional charts (varga)
- Muhūrta (electional astrology)
- Compatibility / synastry
- Progressions or solar arc directions

---

3. Architectural Overview

Aspectarian has four layers. Each has a single, clearly bounded responsibility.





The separation is strict. Layer 1 never calls an LLM. Layer 3 never calls Swiss Ephemeris. Layer 2 is pure assembly — no inference of its own. Layer 4 stores and retrieves; it never transforms.

---

4. Data Model

4.1 User

```
{
  "id": "uuid",
  "name": "string",
  "birth_datetime_utc": "ISO 8601",
  "latitude": float,
  "longitude": float,
  "place_name": "string (optional)",
  "timezone": "string (IANA tz, optional — for display only)",
  "created_at": "ISO 8601"
}
```

v1: one user per installation. v2: multiple users, keyed by OAuth sub.

4.2 NatalChart (cached)

```
{
  "user_id": "uuid",
  "chart_data": { },
  "computed_at": "ISO 8601"
}
```

Natal charts are immutable once computed. Invalidated only if the user edits birth data. Keyed by `user_id`.

4.3 Reading

```
{
  "id": "uuid",
  "user_id": "uuid",
  "window": "day | week | month",
  "ref_start": "YYYY-MM-DD",
  "ref_end": "YYYY-MM-DD",
  "top_contacts": [
    {
      "transit": "string",
      "natal": "string",
      "contact_type": "yuti | drsti",
      "drsti_grade": "string | null",
      "strength": "string"
    }
  ],
  "period_summary": "string",
  "full_report": "string",
  "created_at": "ISO 8601"
}
```

`top_contacts` : the top 3 contacts by score, summarised for lightweight history injection.
`full_report` : full Claude output stored for v2 on-demand retrieval.
`period_summary` : Claude's summary paragraph — injected verbatim into future sessions.

4.4 Session

```
{
  "id": "uuid",
  "user_id": "uuid",
  "reading_id": "uuid | null",
  "system_prompt": "string",
  "conversation_history": [ ],
  "created_at": "ISO 8601",
  "last_active": "ISO 8601",
  "status": "active | closed"
}
```

`reading_id` is set once the session's initial report is saved as a Reading.
`conversation_history` : the full messages array passed to Claude, grows with each turn.

4.5 UserPreferences

```
{
  "user_id": "uuid",
  "theme":           "dark | light",
  "default_window":  "day | week | month",
  "drsti_min_grade": "quarter | half | three_quarter | full",
  "planet_display":  "symbol | abbreviation",
  "name_language":   "sanskrit | english",
  "time_format":     "12h | 24h"
}
```

Defaults: `dark` / `week` / `quarter` / `abbreviation` / `sanskrit` / `24h` .

`planet_display` and `name_language` are display-layer only — they affect the chart panel and UI labels. Claude's output is always IAST regardless. `drsti_min_grade` is applied server-side at contact computation time and passed to `assemble_system_prompt()` .

4a. Location System

4a.1 Dependencies

Package	Role
GeoNames `cities15000.txt`	Bundled city database (~25,000 cities, ~5MB compressed)
`timezonefinder`	Offline lat/lon → IANA timezone conversion
`pytz` or `zoneinfo`	Local birth time → UTC conversion

4a.2 GeoNames SQLite Import

On first launch, if `geonames.db` does not exist, import `cities15000.txt` into a SQLite table:

```
CREATE TABLE IF NOT EXISTS cities (
    geoname_id   INTEGER PRIMARY KEY,
    name         TEXT NOT NULL,
    ascii_name   TEXT NOT NULL,
    latitude     REAL NOT NULL,
    longitude    REAL NOT NULL,
    country_code TEXT NOT NULL,
    admin1_code  TEXT,
    timezone     TEXT NOT NULL,
    population   INTEGER
);
CREATE INDEX idx_cities_ascii ON cities(ascii_name COLLATE NOCASE);
```

Import runs once, takes ~2 seconds, is silent to the user.

4a.3 City Search

```
def search_cities(query: str, limit: int = 10) -> list[dict]:
    """
    Fuzzy prefix search on ascii_name. Returns cities sorted by population desc.
    """
    return db.execute(
        """SELECT name, ascii_name, latitude, longitude, country_code,
                  admin1_code, timezone, population
           FROM cities
           WHERE ascii_name LIKE ?
           ORDER BY population DESC
           LIMIT ?""",
        (f"{query}%", limit)
    ).fetchall()
```

Results displayed as: `{name}, {admin1_code}, {country_code}` (e.g. "Chennai, TN, IN").

4a.4 Birth Time → UTC Conversion

```
from timezonefinder import TimezoneFinder
from zoneinfo import ZoneInfo  # Python 3.9+
from datetime import datetime

tf = TimezoneFinder()

def local_to_utc(date_str: str, time_str: str, lat: float, lon: float) ->
datetime:
    """Convert local birth time to UTC using coordinates."""
    tz_name = tf.timezone_at(lat=lat, lng=lon)
    local_dt = datetime.fromisoformat(f"{date_str}T{time_str}")
    aware_dt  = local_dt.replace(tzinfo=ZoneInfo(tz_name))
    return aware_dt.astimezone(ZoneInfo("UTC"))
```

The timezone string (e.g. `"Asia/Kolkata"`) is stored on the User record for display. The birth_datetime_utc stored in the database is always UTC.

4b. Session Manager

`session/manager.py` is the core conversation engine. It owns the full lifecycle of a session: assembling context, streaming the initial report, handling follow-up Q&A, and persisting readings. All LLM calls go through the injected `LLMClient` instance — never through the Anthropic SDK directly.

4b.0 Supporting Types and Functions

```
from dataclasses import dataclass

@dataclass
class ReportData:
    transits:    TransitSnapshot
    contacts:    list[Contact]
    moon_events: list[NakshatraEvent] | None   # non-None for day reports only

@dataclass
class NakshatraEvent:
    time_utc:   datetime
    nakshatra:  str
    pada:       int
    sign:       str
    event:      str | None   # "nakshatra_change" | "sign_change" | None


def window_dates(window: str, ref_date: date) -> tuple[date, date]:
    """Return (ref_start, ref_end) for the given window and reference date."""
    if window == 'day':
        return ref_date, ref_date
    elif window == 'week':
        return ref_date, ref_date + timedelta(days=6)
    elif window == 'month':
        # Start of the month containing ref_date to end of that month
        start = ref_date.replace(day=1)
        end   = (start + timedelta(days=32)).replace(day=1) - timedelta(days=1)
        return start, end
    raise ValueError(f"Unknown window: {window}")


def compute_report_data(
    natal:     NatalChart,
    window:    str,
```

```
    ref_start: date,
    ref_end:   date,
    prefs:     UserPreferences,
) -> ReportData:
    """Dispatcher — delegates to the appropriate window-specific function."""
    if window == 'day':
        from reports.day import compute_day_report
        return compute_day_report(natal, ref_start, prefs)
    elif window == 'week':
        from reports.week import compute_week_report
        return compute_week_report(natal, ref_start, prefs)
    elif window == 'month':
        from reports.month import compute_month_report
        return compute_month_report(natal, ref_start, prefs)
    raise ValueError(f"Unknown window: {window}")


def filter_by_grade(contacts: list[Contact], min_grade: str) -> list[Contact]:
    GRADE_ORDER = {"quarter": 1, "half": 2, "three_quarter": 3, "full": 4}
    min_val = GRADE_ORDER.get(min_grade, 1)
    return [
        c for c in contacts
        if c.contact_type == 'yuti'                                    # yuti
always included
        or GRADE_ORDER.get(c.drsti_grade, 0) >= min_val
    ]
```

4b.1 start_session()

```
async def start_session(
    user:       User,
    natal:      NatalChart,
    prefs:      UserPreferences,
    window:     str,
    ref_date:   date,
    llm_client: LLMClient,
) -> AsyncGenerator[dict, None]:
    """
    Assembles context, streams NDJSON report via LLMClient, saves reading.
    Yields typed dicts (NDJSON objects + final session_ready object).
```

```
    """
    # 1. Compute transit window
    ref_start, ref_end = window_dates(window, ref_date)
    report_data = compute_report_data(natal, window, ref_start, ref_end, prefs)

    # 2. Fetch reading history
    history = await db.get_history(user.id, n=settings.HISTORY_MAX_READINGS)

    # 3. Assemble system prompt
    system_prompt = assemble_system_prompt(
        natal       = natal,
        transits    = report_data.transits,
        contacts    = report_data.contacts,
        symbol_map  = load_symbol_map(),
        history     = history,
        window      = window,
        ref_start   = ref_start,
        ref_end     = ref_end,
        moon_events = report_data.moon_events,
    )

    # 4. Create session record
    session_id = await db.create_session(user.id, system_prompt)

    # 5. Stream NDJSON via LLMClient (never calls Anthropic SDK directly)
    buffer = ''
    report_objects = []
    messages = [{"role": "user",
                 "content": f"Please generate my {window} transit report."}]

    async for chunk in llm_client.stream(system_prompt, messages,
                                         settings.CLAUDE_MAX_TOKENS):
        buffer += chunk
        objects, buffer = parse_stream(buffer)
        for obj in objects:
            report_objects.append(obj)
            yield obj                          # SSE route re-emits as typed
event

    # Flush remaining buffer
```

```
        if buffer.strip():
            try:
                obj = json.loads(buffer.strip())
                report_objects.append(obj)
                yield obj
            except json.JSONDecodeError:
                logger.warning("Unflushed NDJSON buffer could not be parsed")

        # 6. Persist reading
        period_summary = extract_period_summary(report_objects)
        top_contacts   = extract_top_contacts(report_data.contacts, n=3)
        full_report    = reconstruct_report(report_objects)

        reading_id = await db.save_reading(
            user_id        = user.id,
            window         = window,
            ref_start      = ref_start,
            ref_end        = ref_end,
            top_contacts   = [c.to_context_dict() for c in top_contacts],
            period_summary = period_summary,
            full_report    = full_report,
        )

        # 7. Update session
        await db.update_session(
            session_id           = session_id,
            reading_id           = reading_id,
            conversation_history = [{"role": "assistant", "content": full_report}],
        )

        yield {"type": "session_ready", "session_id": session_id, "reading_id":
    reading_id}
```

4b.2 send_message()

```
    async def send_message(
        session_id: str,
        message:    str,
        llm_client: LLMClient,
    ) -> AsyncGenerator[str, None]:
```

```
    """
    Appends user message, streams plain prose response via LLMClient.
    Yields raw text tokens. Updates conversation history on completion.
    """
    session = await db.get_session(session_id)
    if not session or session.status == 'closed':
        raise ValueError(f"Session {session_id} not found or closed")

    history = session.conversation_history
    history.append({"role": "user", "content": message})

    full_response = ''
    async for token in llm_client.stream(session.system_prompt, history,
                                         settings.CLAUDE_MAX_TOKENS):
        full_response += token
        yield token                              # SSE route re-emits as token
event

    history.append({"role": "assistant", "content": full_response})
    await db.update_session_history(session_id, history)
```

4b.3 `reconstruct_report()`

Converts NDJSON objects to a human-readable plain text string. Used for:

- Storing `full_report` in the Reading record
- CLI terminal display (see §4b.4)

```
def reconstruct_report(report_objects: list[dict]) -> str:
    parts = []
    for obj in report_objects:
        if obj['type'] == 'contact':
            parts.append(
                f"{obj['headline']}\n\n"
                f"{obj['body']}\n\n"
                f"{obj['guna']}\n"
                f"Tone: {obj['tone']}"
            )
        elif obj['type'] == 'period_summary':
            parts.append(f"Period summary\n\n{obj['text']}")
```

```
        elif obj['type'] == 'lunar_weather':
            parts.append(
                f"Lunar weather\n\n"
                f"{obj['nakshatra_start']} → {obj['nakshatra_end']}\n"
                f"{obj['note']}"
            )
        elif obj['type'] == 'history_note':
            parts.append(obj['text'])
    return "\n\n——\n\n".join(parts)
```

4b.4 CLI Terminal Display

The CLI renders the NDJSON stream to the terminal using `reconstruct_report()` progressively — each object is formatted and printed as it arrives rather than waiting for the full report:

```
async for obj in session_manager.start_session(...):
    if obj['type'] == 'contact':
        print(f"\n{obj['headline']}")
        print(f"{obj['body']}")
        print(f"  {obj['guna']}")
        print(f"  Tone: {obj['tone']}\n")
    elif obj['type'] == 'period_summary':
        print("—" * 60)
        print(f"Period summary\n\n{obj['text']}\n")
    elif obj['type'] == 'lunar_weather':
        print(f"Lunar weather\n{obj['note']}\n")
    elif obj['type'] == 'session_ready':
        print("—" * 60)
        print("Report complete. Ask a question or type 'exit'.\n")
```

4c. Report Window Sampling

`reports/day.py` , `week.py` , `month.py` each implement `compute_report_data()` for their window type. All return a `ReportData` object containing `transits` , `contacts` , and optionally `moon_events` .

4c.1 Day Report

```
def compute_day_report(natal: NatalChart, ref_date: date,
                       prefs: UserPreferences) -> ReportData:
    dt = datetime(ref_date.year, ref_date.month, ref_date.day,
                  settings.TRANSIT_SAMPLE_HOUR_UTC, tzinfo=UTC)
    transits = compute_transits(dt)
    contacts = find_all_contacts(transits, natal)
    contacts = filter_by_grade(contacts, prefs.drsti_min_grade)
    moon_events = compute_moon_nakshatra_events(ref_date)
    return ReportData(transits=transits, contacts=contacts,
moon_events=moon_events)
```

4c.2 Week Report

Three snapshots: day 1, day 4, day 7 of the week. A contact is **active** if present in ≥ 2 of 3 snapshots. Moon summary computed daily.

```
def compute_week_report(natal: NatalChart, ref_start: date,
                        prefs: UserPreferences) -> ReportData:
    sample_days = [ref_start,
                   ref_start + timedelta(days=3),
                   ref_start + timedelta(days=6)]
    all_snapshots = []
    for day in sample_days:
        dt = datetime(day.year, day.month, day.day,
                      settings.TRANSIT_SAMPLE_HOUR_UTC, tzinfo=UTC)
        transits = compute_transits(dt)
        contacts = find_all_contacts(transits, natal)
        all_snapshots.append(contacts)

    # Contact is active if the (transit, natal) pair appears in ≥2 snapshots
    active = majority_contacts(all_snapshots, threshold=2)
    active = filter_by_grade(active, prefs.drsti_min_grade)

    # Use mid-week snapshot as the canonical transit positions for the report
    mid_transits = compute_transits(
        datetime(sample_days[1].year, sample_days[1].month,
                 sample_days[1].day, settings.TRANSIT_SAMPLE_HOUR_UTC,
tzinfo=UTC)
    )
```

```
    return ReportData(transits=mid_transits, contacts=active, moon_events=None)


def majority_contacts(snapshots: list[list[Contact]],
                      threshold: int) -> list[Contact]:
    """Return contacts active in ≥ threshold snapshots.
    Uses the mid-point snapshot's Contact object for grade/degree data
    when available; falls back to any snapshot's data otherwise.
    """
    counter = defaultdict(int)
    mid_map = {}   # prefer mid-point snapshot (index len//2)
    any_map = {}
    mid_idx = len(snapshots) // 2
    for i, snapshot in enumerate(snapshots):
        for c in snapshot:
            key = (c.transit, c.natal)
            counter[key] += 1
            any_map[key] = c
            if i == mid_idx:
                mid_map[key] = c
    return [mid_map.get(k, any_map[k])
            for k, count in counter.items() if count >= threshold]
```

4c.3 Month Report

Four weekly snapshots. Only **slow planets** (Jupiter, Saturn, Rāhu, Ketu, Uranus, Neptune, Pluto) produce individual contact entries. Fast planets are excluded from the contact list — they are noted in the system prompt as a group for Claude to summarise thematically.

Retrograde **stations** (planet turning retrograde or direct) are detected if they fall within the month and flagged in the system prompt.

```
SLOW_PLANETS = {"Jupiter", "Saturn", "Rahu", "Ketu", "Uranus", "Neptune",
"Pluto"}

def compute_month_report(natal: NatalChart, ref_start: date,
                         prefs: UserPreferences) -> ReportData:
    sample_days = [ref_start + timedelta(weeks=i) for i in range(4)]
    all_snapshots = []
    for day in sample_days:
```

```
            dt = datetime(day.year, day.month, day.day,
                          settings.TRANSIT_SAMPLE_HOUR_UTC, tzinfo=UTC)
            transits = compute_transits(dt)
            contacts = find_all_contacts(transits, natal)
            all_snapshots.append(contacts)

        # Only slow-planet contacts, active in ≥2 of 4 snapshots
        active = majority_contacts(all_snapshots, threshold=2)
        active = [c for c in active if c.transit in SLOW_PLANETS]
        active = filter_by_grade(active, prefs.drsti_min_grade)

        # Mid-month transits as canonical positions
        mid_transits = compute_transits(
            datetime(sample_days[1].year, sample_days[1].month,
                     sample_days[1].day, settings.TRANSIT_SAMPLE_HOUR_UTC,
    tzinfo=UTC)
        )

        return ReportData(transits=mid_transits, contacts=active, moon_events=None)
```

4d. Daśā System (v2)

Daśās determine the temporal activation of chart energies. Without them, Aspectarian identifies *what* is astrologically active but not *when* it is most likely to crystallise into experience. The daśā period is the container within which transits operate — a transit involving the current daśā lord is qualitatively different from the same transit in a neutral period.

Aspectarian implements **Viṃśottarī daśā** — the most widely used system in Parāśaran Jyotiṣa — as a three-phase v2 addition.

4d.1 Viṃśottarī Calculation

Viṃśottarī is calculated from the Moon's nakṣatra at birth. The full cycle is 120 years across 9 planetary periods in fixed sequence:

Lord	Years	Lord	Years
Ketu	7	Rāhu	18
Śukra	20	Guru	16
Sūrya	6	Śani	19
Candra	10	Budha	17
Maṅgala	7	*(cycle repeats)*	

The balance of daśā at birth is determined by the Moon's position within its nakṣatra (each spans 13°20′). The fraction of the nakṣatra elapsed determines what fraction of that lord's period has been consumed.

```
def compute_vimshottari(moon_lon: float, birth_dt: datetime) -> DasaPeriod:
    """
    Compute current mahādaśā, antardaśā, and pratyantardaśā from Moon longitude
    and birth datetime. Returns active periods with exact date ranges.
    """
```

Birth time sensitivity: The Moon moves ~13° per day — roughly one nakṣatra every 13 hours. An error of one hour in birth time produces approximately 32′ of Moon position error, which can shift the daśā balance by several months. Birth time accuracy to within 15 minutes is recommended for reliable daśā results. If birth time is unknown, daśā readings must be clearly caveated.

4d.2 Integration Levels

Phase 1 — Daśā awareness (v2.1): Compute the current mahādaśā, antardaśā, and pratyantardaśā. Inject them into the context builder as a `<dasa_period>` XML block. Claude references them when interpreting transits, noting when a transit involves the current daśā lord. No new report window needed.

Phase 2 — Daśā report window (v2.2): A fourth window type (`dasa`) alongside day / week / month. Interprets the current mahādaśā / antardaśā period in depth — the lord's significations, psychological and spiritual themes, and what the period tends to activate. Not transit-dependent. Can be combined with a transit report in the same session.

Phase 3 — Daśā-transit synthesis (v2.3): Transit contacts involving the current mahādaśā or antardaśā lord receive a score modifier of +2, reflecting that transits become significantly more powerful when the planet involved is simultaneously activated by daśā. The modifier is applied in `score_contact()` and noted in the NDJSON contact object.

4d.3 `<dasa_period>` Context Block

Added to the system prompt in v2.1, positioned after `<active_contacts>` and before `<retrograde_planets>`. Omitted in v1.

```
<dasa_period>
{
  "maha_dasa":       { "lord": "Rahu",    "start": "2019-03-15", "end": "2037-
03-15" },
  "antar_dasa":      { "lord": "Jupiter", "start": "2025-09-12", "end": "2028-
05-20" },
  "pratyantar_dasa": { "lord": "Saturn",  "start": "2026-02-14", "end": "2026-
10-08" },
  "balance_at_birth": { "dasa_lord": "Rahu", "years_remaining": 12.4 }
}
</dasa_period>
```

Claude uses this to note when an active transit involves the current daśā lord, contextualise the period's overall flavour when answering timing questions, and flag approaching daśā changes if they fall within the report window.

4d.4 New Module

`engine/dasa.py` — added in v2.1:

```
def compute_vimshottari(moon_lon: float, birth_dt: datetime) -> DasaPeriod
def get_active_periods(birth_dt: datetime, query_dt: datetime) -> DasaPeriod
def dasa_lord_modifier(contact: Contact, dasa: DasaPeriod) -> int
# Returns +2 if contact.transit is mahādaśā or antardaśā lord, else 0
```

5.1 Astronomical Parameters

- **Ayanāṃśa:** Lahiri (Chitrapakṣa), `swe.SIDM_LAHIRI`
- **Coordinate system:** Sidereal ecliptic longitude
- **House system:** Whole-sign. Bhāva = rāśi. The sign a graha occupies is its house. No cusp calculation for bhāva assignment.
- **Lagna:** Computed from `swe.houses()` → converted to sidereal → sign determines lagna sign

5.2 Grahas Tracked

Graha	Sanskrit	Category	Priority
Sun	Sūrya	Fast	Secondary
Moon	Candra	Fast	Primary (nakṣatra timing)
Mercury	Budha	Fast	Secondary
Venus	Śukra	Fast	Secondary
Mars	Maṅgala	Fast	Secondary
Jupiter	Guru	Slow	Primary
Saturn	Śani	Slow	Primary
Rāhu	Rāhu	Slow	Primary
Ketu	Ketu	Slow	Primary
Uranus	—	Slow	Primary
Neptune	—	Slow	Primary
Pluto	—	Slow	Primary

Ketu: Always derived as `(rahu_lon + 180.0) % 360.0`. Never fetched separately from Swiss Ephemeris.

5.3 Retrograde Detection

For each graha (excluding Rāhu, Ketu, Sun, Moon — these do not retrograde in Jyotiṣa):

```
flags = swe.FLG_SIDEREAL | swe.FLG_SPEED
result = swe.calc_ut(jd, planet, flags)
is_retrograde = result[0][3] < 0  # negative daily motion = retrograde
```

Retrograde status is passed to the Context Builder and influences tone in the system prompt.

5.4 Contact System — Vedic Dṛṣṭi + Yuti

Aspectarian uses the classical Parāśaran dṛṣṭi system. Contacts are sign-based, not degree-based. There are no orbs. Two contact types are distinguished: **yuti** (co-presence in the same sign) and **dṛṣṭi** (directional aspect from a distance). These are separate saṃbandha categories and must not be conflated.

House Count Calculation

```
house_count = ((natal_sign_index - transit_sign_index) % 12) + 1
# sign_index: Aries=0, Taurus=1, ..., Pisces=11
# Result: 1–12, counting forward through the zodiac from the transit planet
```

Yuti (Co-presence)

When `house_count == 1`: the transit planet occupies the same sign as the natal planet. This is yuti — not a dṛṣṭi. Both planets mutually condition each other's field. The contact is recorded as `contact_type: "yuti"` with no grade.

Self-contacts are excluded: if transit graha == natal graha, skip.

Base Dṛṣṭi (All Planets)

House count	Dṛṣṭi	Grade	Fraction
7	Saptama dṛṣṭi	Full	4/4
4	Caturtha dṛṣṭi	Three-quarter	3/4
8	Aṣṭama dṛṣṭi	Three-quarter	3/4
5	Pañcama dṛṣṭi	Half	2/4
9	Navama dṛṣṭi	Half	2/4
3	Tṛtīya dṛṣṭi	Quarter	1/4
10	Daśama dṛṣṭi	Quarter	1/4
2, 6, 11, 12	—	None	—

Special Dṛṣṭi (Upgrades Grade to Full)

Graha	Special houses (full dṛṣṭi)	Base grade without special
Maṅgala	4th, 8th	Three-quarter → Full
Guru	5th, 9th	Half → Full
Śani	3rd, 10th	Quarter → Full
Rāhu	5th, 9th	Half → Full
Ketu	5th, 9th	Half → Full

Rāhu and Ketu follow Parāśara's assignment (5th and 9th, same as Guru). These are powerful contacts in practice and treated as full dṛṣṭi.

Implementation Sketch

```
SPECIAL_DRSTI = {
    "Mars":    [4, 8],
    "Jupiter": [5, 9],
    "Saturn":  [3, 10],
    "Rahu":    [5, 9],
    "Ketu":    [5, 9],
}

BASE_GRADES = {7: "full", 4: "three_quarter", 8: "three_quarter",
               5: "half", 9: "half", 3: "quarter", 10: "quarter"}

def compute_contact(transit_graha, transit_sign, natal_graha, natal_sign):
    if transit_graha == natal_graha:
        return None  # self-contact excluded
    hc = ((natal_sign - transit_sign) % 12) + 1
    if hc == 1:
        return {"contact_type": "yuti", "house_count": 1,
                "drsti_grade": None, "is_special": False}
    if hc not in BASE_GRADES:
        return None  # no contact
    grade = BASE_GRADES[hc]
    is_special = hc in SPECIAL_DRSTI.get(transit_graha, [])
    if is_special:
        grade = "full"
    return {"contact_type": "drsti", "house_count": hc,
            "drsti_grade": grade, "is_special": is_special}
```

Minimum Grade Filter

By default all grades are included. The `DRSTI_MIN_GRADE` config parameter can raise the floor to `"half"` or `"three_quarter"` to reduce noise in busy charts.

5.5 Contact Strength Scoring

Base score by contact type and grade:

Contact type	Grade	Base score
Yuti	—	5
Dṛṣṭi	Full	4
Dṛṣṭi	Three-quarter	3
Dṛṣṭi	Half	2
Dṛṣṭi	Quarter	1

Modifiers:

Factor	Points
Slow transit planet (Guru, Śani, Rāhu, Ketu, Uranus, Neptune, Pluto)	+2
Contact is special dṛṣṭi	+1
Transit planet is retrograde	+1
Natal planet is lagna lord or ātmakāraka	+1

Score → strength label: `strong` (7+), `moderate` (4–6), `weak` (1–3).

5.6 Moon's Nakṣatra Tracking (Day Reports)

For day reports, the Moon's nakṣatra and pāda are computed at the start, midpoint, and end of the day (00:00, 12:00, 23:59 UTC). If a sign or nakṣatra change occurs within the window, it is flagged and passed to the Context Builder for inclusion in the system prompt.

Topocentric correction (v2): All planetary longitudes in v1 are geocentric — computed from Earth's centre, which is sufficient for all grahas. The Moon is the exception: at ~384,000 km distance, the topocentric correction (observer's actual position on Earth's surface) can shift its longitude by up to ~1°. For dṛṣṭi this only matters when the Moon sits within ~1° of a sign boundary. v1 uses geocentric Moon positions and treats this as a known limitation. v2 will add optional topocentric correction via `swe.FLG_TOPOCTR`, which requires the user's **current** latitude, longitude, and altitude — not their birth location.

5.7 Computation Output Schema

```
{
  "natal_chart": {
```

```
    "lagna": { "sign": "str", "degree": float, "nakshatra": "str", "pada": int
},
    "planets": {
      "<GrahaName>": {
        "sign": "str", "degree": float, "nakshatra": "str", "pada": int
      }
    }
  },
  "transits": {
    "<GrahaName>": {
      "sign": "str", "degree": float, "nakshatra": "str", "pada": int,
      "retrograde": bool
    }
  },
  "active_contacts": [
    {
      "transit": "str",
      "natal": "str",
      "contact_type": "yuti | drsti",
      "house_count": int,
      "drsti_grade": "full | three_quarter | half | quarter | null",
      "is_special": bool,
      "strength": "strong | moderate | weak",
      "score": int,
      "transit_retrograde": bool
    }
  ],
  "moon_nakshatra_events": [
    { "time_utc": "ISO 8601", "nakshatra": "str", "pada": int, "sign": "str" }
  ],
  "retrograde_planets": ["Saturn", "..."],
  "window": "day | week | month",
  "reference_period": { "start": "YYYY-MM-DD", "end": "YYYY-MM-DD" }
}
```

---

6. Layer 2 — Context Builder

The Context Builder transforms computation output into a Claude system prompt. It is deterministic, stateless, and has no LLM calls of its own.

Formatting principle: All data blocks are JSON wrapped in XML tags. All instruction blocks are plain text. XML tags give Claude unambiguous structural boundaries; JSON eliminates formatting ambiguity in data.

6.1 System Prompt Structure

Blocks are assembled in this order. Conditional blocks are omitted when their condition is false.

```
[ROLE — plain text]

<framework>
  symbol_map.json contents (verbatim JSON)
</framework>

<natal_chart>
  NatalChart JSON
</natal_chart>

<active_transits>
  TransitSnapshot JSON
</active_transits>

<active_contacts>
  JSON array, sorted by score descending
</active_contacts>

<dasa_period>                    ← v2.1+; omitted in v1
  JSON object (mahādaśā, antardaśā, pratyantardaśā, balance at birth)
</dasa_period>

<retrograde_planets>             ← omitted if no retrogrades
  JSON array of strings
</retrograde_planets>

<reading_history>                ← omitted if no past readings
  JSON array, oldest first
</reading_history>

<moon_nakshatra_events>          ← day reports only
  JSON array
```

```
</moon_nakshatra_events>

<report_window>
  JSON object
</report_window>

[RETROGRADE INSTRUCTION — plain text, conditional on retrograde_planets]

[INTERPRETATION INSTRUCTIONS — plain text]
```

6.2 Role Block (plain text)

```
You are an expert Vedic astrologer practicing within the Jyotiṣa tradition,
augmented by the guṇa-consciousness framework in <framework>.
Your role is to interpret the transit chart for the user and to answer their
follow-up questions within a single session. You hold the full astrological
context for this session in the XML blocks above. You do not have access to
any other information about the user beyond what is provided here.
```

6.3 `<framework>` Block

The full contents of `symbol_map.json` injected verbatim as JSON inside the tag.

```
<framework>
{
  "meta": { ... },
  "graha": { ... },
  "rasi": { ... },
  "bhava": { ... },
  "aspects": { ... },
  "guna_system": { ... },
  "correlations": { ... },
  "interpretation_rules": { ... }
}
</framework>
```

6.4 `<natal_chart>` Block

The full NatalChart from computation output. No fields omitted.

```
<natal_chart>
{
  "lagna": {
    "sign": "Pisces",
    "degree": 23.19,
    "nakshatra": "Revatī",
    "pada": 2
  },
  "planets": {
    "Sun":     { "sign": "Aquarius",    "degree": 9.00,  "nakshatra":
"Śatabhiṣaj",    "pada": 1 },
    "Moon":    { "sign": "Aquarius",    "degree": 14.97, "nakshatra":
"Śatabhiṣaj",    "pada": 3 },
    "Mercury": { "sign": "Aquarius",    "degree": 27.06, "nakshatra":
"Pūrvabhādrapadā","pada": 3 },
    "Venus":   { "sign": "Sagittarius", "degree": 23.70, "nakshatra":
"Pūrvāṣāḍhā",   "pada": 4 },
    "Mars":    { "sign": "Capricorn",   "degree": 28.14, "nakshatra":
"Dhaniṣṭhā",     "pada": 2 },
    "Jupiter": { "sign": "Scorpio",     "degree": 3.78,  "nakshatra":
"Anurādhā",      "pada": 1 },
    "Saturn":  { "sign": "Cancer",      "degree": 10.30, "nakshatra": "Puṣya",
      "pada": 3 },
    "Rahu":    { "sign": "Taurus",      "degree": 14.30, "nakshatra": "Rohiṇī",
       "pada": 2 },
    "Ketu":    { "sign": "Scorpio",     "degree": 14.30, "nakshatra":
"Anurādhā",      "pada": 2 },
    "Uranus":  { "sign": "Taurus",      "degree": 24.64, "nakshatra":
"Mṛgaśīrṣa",    "pada": 1 },
    "Neptune": { "sign": "Virgo",       "degree": 17.26, "nakshatra": "Hasta",
      "pada": 3 },
    "Pluto":   { "sign": "Cancer",      "degree": 18.63, "nakshatra": "Āśleṣā",
       "pada": 1 }
  }
}
</natal_chart>
```

6.5 `<active_transits>` Block

Current transit positions. Includes `retrograde` flag on each planet. Does not include contacts (those are in `<active_contacts>`).

```
<active_transits>
{
  "Sun":     { "sign": "Pisces",    "degree": 28.16, "nakshatra": "Revatī",
       "pada": 4, "retrograde": false },
  "Moon":    { "sign": "Capricorn", "degree": 21.89, "nakshatra": "Śravaṇa",
      "pada": 4, "retrograde": false },
  "Mercury": { "sign": "Pisces",    "degree": 1.90,  "nakshatra":
"Pūrvabhādrapadā",   "pada": 4, "retrograde": false },
  "Venus":   { "sign": "Aries",     "degree": 21.28, "nakshatra": "Bharaṇī",
      "pada": 3, "retrograde": false },
  "Mars":    { "sign": "Pisces",    "degree": 7.70,  "nakshatra":
"Uttarabhādrapadā",  "pada": 2, "retrograde": false },
  "Jupiter": { "sign": "Gemini",    "degree": 22.47, "nakshatra": "Punarvasu",
      "pada": 1, "retrograde": false },
  "Saturn":  { "sign": "Pisces",    "degree": 12.72, "nakshatra":
"Uttarabhādrapadā",  "pada": 3, "retrograde": true  },
  "Rahu":    { "sign": "Aquarius",  "degree": 12.58, "nakshatra": "Śatabhiṣaj",
       "pada": 2, "retrograde": true  },
  "Ketu":    { "sign": "Leo",       "degree": 12.58, "nakshatra":
"Pūrvaphalgunī",     "pada": 2, "retrograde": true  },
  "Uranus":  { "sign": "Taurus",    "degree": 5.06,  "nakshatra": "Kṛttikā",
      "pada": 3, "retrograde": false },
  "Neptune": { "sign": "Pisces",    "degree": 8.40,  "nakshatra":
"Uttarabhādrapadā",  "pada": 2, "retrograde": false },
  "Pluto":   { "sign": "Capricorn", "degree": 11.15, "nakshatra": "Śravaṇa",
      "pada": 1, "retrograde": false }
}
</active_transits>
```

6.6 `<active_contacts>` Block

JSON array of contact objects, sorted by `score` descending. Contains **only computation fields** — `headline`, `body`, `guna`, `consciousness`, and `tone` are Claude's output and must not appear here.

```
<active_contacts>
[
  {
```

```
    "transit":          "Rahu",
    "natal":            "Moon",
    "contact_type":     "yuti",
    "drsti_grade":      null,
    "is_special":       false,
    "house_count":      1,
    "score":            8,
    "strength":         "strong",
    "transit_retrograde": true
  },
  {
    "transit":          "Pluto",
    "natal":            "Saturn",
    "contact_type":     "drsti",
    "drsti_grade":      "full",
    "is_special":       false,
    "house_count":      7,
    "score":            7,
    "strength":         "strong",
    "transit_retrograde": false
  }
]
</active_contacts>
```

6.7 `<retrograde_planets>` Block

JSON array of transit planet names that are currently retrograde. **Omitted entirely** when no planets are retrograde.

```
<retrograde_planets>
["Saturn", "Rahu", "Ketu"]
</retrograde_planets>
```

6.8 `<reading_history>` Block

JSON array of past Reading summaries, ordered **oldest-first** so Claude reads them chronologically. **Omitted entirely** if the user has no past readings.

- Maximum N readings, configured by `HISTORY_MAX_READINGS` (default 5)
- `full_report` is **not** included — summary fields only

- `period_summary` is included verbatim (Claude wrote it; injecting it back is reliable)

```
<reading_history>
[
  {
    "ref_start": "2026-03-15",
    "ref_end":   "2026-04-14",
    "window":    "month",
    "top_contacts": [
      { "transit": "Neptune", "natal": "Moon",   "contact_type": "yuti",
"drsti_grade": null,   "strength": "strong"   },
      { "transit": "Jupiter", "natal": "Venus",  "contact_type":
"drsti","drsti_grade": "full",  "strength": "moderate" }
    ],
    "period_summary": "The dissolution of habitual emotional patterns continues,
met by Jupiter's expansive influence in the relational domain. A month for
releasing attachment to familiar feeling-states."
  },
  {
    "ref_start": "2026-04-05",
    "ref_end":   "2026-04-11",
    "window":    "week",
    "top_contacts": [
      { "transit": "Saturn", "natal": "Moon",   "contact_type": "drsti",
"drsti_grade": "full", "strength": "strong" },
      { "transit": "Pluto",  "natal": "Saturn", "contact_type": "drsti",
"drsti_grade": "full", "strength": "strong" }
    ],
    "period_summary": "Sustained structural pressure on the lunar field from two
slow planets. Inner consolidation is more available than outward movement."
  }
]
</reading_history>
```

6.9 `<moon_nakshatra_events>` Block

Day reports only. Omitted for week and month reports. JSON array of nakṣatra snapshots and any intra-day transitions.

```
<moon_nakshatra_events>
[
  { "time_utc": "2026-04-12T00:00:00+00:00", "nakshatra": "Uttarabhādrapadā",
"pada": 3, "sign": "Pisces"  },
  { "time_utc": "2026-04-12T18:42:00+00:00", "nakshatra": "Revatī",
"pada": 1, "sign": "Pisces"  }
]
</moon_nakshatra_events>
```

If no sign or nakṣatra change occurs during the day, the array contains only the start-of-day snapshot.

6.10 `<report_window>` Block

```
<report_window>
{
  "window": "week",
  "start":  "2026-04-12",
  "end":    "2026-04-18"
}
</report_window>
```

6.11 Retrograde Instruction Block (plain text, conditional)

Injected as plain text **only when `<retrograde_planets>` is present**. Placed immediately after `<report_window>` and before the interpretation instructions.

```
RETROGRADE RULE: When interpreting contacts involving a retrograde transit
planet,
intensify the themes of that graha and add an inward, revisionary quality to the
interpretation. Retrograde energy turns outward expression inward — the themes
resurface from previous cycles and demand internal processing rather than
external
action. Mark retrograde planets with ℞ in your output.
```

6.12 Interpretation Instructions Block (plain text)

(Unchanged — see §6.12 for full content)

```
OUTPUT FORMAT — INITIAL REPORT ONLY:
```

```
Generate the transit report as newline-delimited JSON (NDJSON).
Each line is one complete JSON object. No markdown, no prose headers,
no blank lines between objects. Emit objects in this order:
  1. One {"type":"contact",...} object per active contact, strongest first
  2. One {"type":"period_summary",...} object
  3. One {"type":"lunar_weather",...} object  (day reports only)
  4. One {"type":"history_note",...} object   (only if <reading_history> is
present)

For follow-up questions (all turns after the first), respond in plain prose.
Do not use NDJSON for follow-up answers.

_____________________________________________________
NDJSON SCHEMAS
_____________________________________________________

Contact object:
{
  "type":               "contact",
  "transit":            "<GrahaName>",
  "natal":              "<GrahaName>",
  "contact_type":       "yuti" | "drsti",
  "drsti_grade":        "full" | "three_quarter" | "half" | "quarter" | null,
  "is_special":         true | false,
  "house_count":        <int 1–12>,
  "score":              <int>,
  "strength":           "strong" | "moderate" | "weak",
  "transit_retrograde": true | false,
  "headline":           "<string, ≤20 words>",
  "body":               "<string, 3–5 sentences>",
  "guna":               "<string, one line>",
  "consciousness":      "<string, one line, or null>",
  "tone":               "challenging" | "supportive" | "neutral" | "mixed"
}

Period summary object:
{
  "type": "period_summary",
  "text": "<string, 3–6 sentences>"
}
```

```
Lunar weather object (day reports only):
{
  "type":            "lunar_weather",
  "nakshatra_start": "<nakṣatra name> pāda <n>",
  "nakshatra_end":   "<nakṣatra name> pāda <n>",
  "changes":         "<sign/nakṣatra changes during the day, or null>",
  "note":            "<string, 2 sentences on the lunar quality of the day>"
}

History note object (only if <reading_history> is present):
{
  "type": "history_note",
  "text": "<string, 1 sentence noting continuity with past readings>"
}

_____________________________________________________
CONTACT TYPE RULES — apply to all headline, body, and guna fields:

  YUTI (contact_type = "yuti"):
  The transit planet occupies the same sign as the natal planet — field-sharing,
  not aspect. Interpret as immersion: the themes do not arrive from outside,
they
  become the texture of experience itself. Use language like "inhabits",
"suffuses",
  "is woven into". Never use "aspects" or "opposes" for yuti.

  DṚṢṬI (contact_type = "drsti"):
  The transit planet casts its gaze from a distance. Modulate intensity by
grade:
    - Full (especially special dṛṣṭi): direct, sustained, high-intensity gaze
    - Three-quarter: strong but oblique pressure
    - Half: moderate, often felt as background resonance
    - Quarter: subtle, a faint but present quality
  Note special dṛṣṭi (is_special = true) explicitly — these are particularly
potent.
  Use language like "aspects", "gazes upon", "casts its influence".

LANGUAGE RULES (apply to all text fields):
  - Use IAST for all Sanskrit terms
```

```
  - Tone is warm, precise, and non-fatalistic
  - Interpret the chart holistically — note when contacts reinforce or
contradict
  - Do not use Western sun-sign astrology concepts or terminology
  - Keep all string values on a single line (no literal newlines inside JSON
strings)
```

6.13 Messages Array

```
messages = [
    { "role": "user", "content": f"Please generate my {window} transit report."
}
]
```

The full conversation history is appended on each subsequent turn. The system prompt never changes within a session.

6.14 Assembly Function

```
def assemble_system_prompt(
    natal:        NatalChart,
    transits:     TransitSnapshot,
    contacts:     list[Contact],
    symbol_map:   dict,
    history:      list[Reading],
    window:       str,
    ref_start:    date,
    ref_end:      date,
    moon_events:  list[NakshatraEvent] | None = None,
) -> str:
    parts = []
    parts.append(ROLE_BLOCK)
    parts.append(wrap_xml("framework",  json.dumps(symbol_map,
ensure_ascii=False, indent=2)))
    parts.append(wrap_xml("natal_chart", json.dumps(natal,
ensure_ascii=False, indent=2)))
    parts.append(wrap_xml("active_transits", json.dumps(transits,
ensure_ascii=False, indent=2)))
    parts.append(wrap_xml("active_contacts",
```

```
            json.dumps([c.to_input_dict() for c in contacts], ensure_ascii=False,
    indent=2)))

        retro = list(dict.fromkeys(c.transit for c in contacts if
    c.transit_retrograde))
        if retro:
            parts.append(wrap_xml("retrograde_planets", json.dumps(retro)))

        if history:
            parts.append(wrap_xml("reading_history",
                json.dumps([r.to_summary_dict() for r in history],
    ensure_ascii=False, indent=2)))

        if window == 'day' and moon_events:
            parts.append(wrap_xml("moon_nakshatra_events",
                json.dumps([e.to_dict() for e in moon_events], ensure_ascii=False,
    indent=2)))

        parts.append(wrap_xml("report_window",
            json.dumps({"window": window, "start": str(ref_start), "end":
    str(ref_end)})))

        if retro:
            parts.append(RETROGRADE_INSTRUCTION)

        parts.append(INTERPRETATION_INSTRUCTIONS)

        return "\n\n".join(parts)


    def wrap_xml(tag: str, content: str) -> str:
        return f"<{tag}>\n{content}\n</{tag}>"
```

`Contact.to_input_dict()` returns only the computation fields — never `headline`, `body`, `guna`, `consciousness`, or `tone`.

`Reading.to_summary_dict()` returns `ref_start`, `ref_end`, `window`, `top_contacts`, and `period_summary` — never `full_report`.

---

7. Layer 3 — Claude API

7.1 API Call

```
response = anthropic.messages.create(
    model="claude-sonnet-4-20250514",
    max_tokens=4096,
    system=system_prompt,
    messages=conversation_history
)
```

7.2 Session Flow

```
1.  User selects window + date → engine computes transits + contacts
2.  Storage fetches natal chart (cached) and reading history
3.  Context Builder assembles system prompt
4.  API call → Claude generates initial transit report
5.  Report displayed in chat UI; chart panel updates with transit overlay
6.  Session saved to storage (Session record, reading_id = null)

--- conversation loop ---
7.  User types follow-up question
8.  Question appended to messages array
9.  API call with same system prompt + full history
10. Claude responds
11. Return to step 7

--- session end ---
12. On exit/close: period_summary and top_contacts extracted from the report,
    Reading record written to storage, Session.reading_id updated
```

7.3 What Claude Does

Claude handles everything interpretive: synthesises contacts into a narrative, applies the guṇa and consciousness framework, weighs strength scores, notes cross-contact reinforcement or contradiction, applies retrograde intensification, references reading history for continuity, and answers follow-up questions on timing, domain, remedies, and deeper symbology.

Claude does not access information outside the system prompt, make predictions about external events, or override the computation layer's data.

8. Storage Layer

8.1 v1 — SQLite

Database file: `aspectarian.db` (path configurable via `DB_PATH`).

```
CREATE TABLE users (
    id TEXT PRIMARY KEY,
    name TEXT NOT NULL,
    birth_datetime_utc TEXT NOT NULL,
    latitude REAL NOT NULL,
    longitude REAL NOT NULL,
    place_name TEXT,
    timezone TEXT,
    created_at TEXT NOT NULL
);

CREATE TABLE natal_charts (
    user_id TEXT PRIMARY KEY REFERENCES users(id),
    chart_data TEXT NOT NULL,   -- JSON blob
    computed_at TEXT NOT NULL
);

CREATE TABLE readings (
    id TEXT PRIMARY KEY,
    user_id TEXT NOT NULL REFERENCES users(id),
    window TEXT NOT NULL,
    ref_start TEXT NOT NULL,
    ref_end TEXT NOT NULL,
    top_contacts TEXT NOT NULL, -- JSON blob
    period_summary TEXT NOT NULL,
    full_report TEXT NOT NULL,
    created_at TEXT NOT NULL
);

CREATE TABLE sessions (
    id TEXT PRIMARY KEY,
    user_id TEXT NOT NULL REFERENCES users(id),
    reading_id TEXT REFERENCES readings(id),
```

```
    system_prompt TEXT NOT NULL,        -- stored at creation; reused for
follow-up turns
    conversation_history TEXT NOT NULL, -- JSON blob
    created_at TEXT NOT NULL,
    last_active TEXT NOT NULL,
    status TEXT NOT NULL DEFAULT 'active'  -- 'active' | 'closed'
);

CREATE TABLE preferences (
    user_id        TEXT PRIMARY KEY REFERENCES users(id),
    theme          TEXT NOT NULL DEFAULT 'dark',
    default_window TEXT NOT NULL DEFAULT 'week',
    drsti_min_grade TEXT NOT NULL DEFAULT 'quarter',
    planet_display TEXT NOT NULL DEFAULT 'abbreviation',
    name_language  TEXT NOT NULL DEFAULT 'sanskrit',
    time_format    TEXT NOT NULL DEFAULT '24h'
);

CREATE TABLE cities (
    geoname_id   INTEGER PRIMARY KEY,
    name         TEXT NOT NULL,
    ascii_name   TEXT NOT NULL,
    latitude     REAL NOT NULL,
    longitude    REAL NOT NULL,
    country_code TEXT NOT NULL,
    admin1_code  TEXT,
    timezone     TEXT NOT NULL,
    population   INTEGER
);
CREATE INDEX idx_cities_ascii ON cities(ascii_name COLLATE NOCASE);
CREATE INDEX idx_sessions_user ON sessions(user_id, last_active DESC);
```

8.2 v2 — PostgreSQL + OAuth

In v2, SQLite is replaced by PostgreSQL. The schema is identical with the following additions: a `providers` table linking OAuth sub to user_id, a `practitioners` table for the Jyotiṣī multi-client model (practitioner_id → client user_ids), and row-level security policies per user.

---

9. Reading History

9.1 What Is Stored

At session end, the system extracts from Claude's output:

- The top 3 contacts by score from the contacts list (structured data, not parsed from text)
- The PERIOD SUMMARY paragraph (identified by the section header in Claude's output)
- The full raw report text

These are written as a Reading record.

9.2 Lightweight History Injection

The Context Builder queries the most recent N readings for the user (default N = 5) and injects a compact summary. Example:

```
READING HISTORY:

15 Mar 2026 (month):
  Top contacts: Neptune yuti Candra (strong), Guru dṛṣṭi Śukra (moderate)
  Summary: The dissolution of habitual emotional patterns continues, met by an
  expansive Jupiter influence in the relational domain. A month for releasing
  attachment to familiar feeling-states.

5 Apr 2026 (week):
  Top contacts: Śani ℞ dṛṣṭi Candra (strong), Pluto dṛṣṭi Śani (strong)
  Summary: Sustained structural pressure on the lunar mind from two slow
planets.
  The karmic weight of Saturn's retrograde gaze intensifies an already
compressed
  feeling of limitation. Inner consolidation is more available than outward
movement.
```

9.3 Full Report Retrieval (v2)

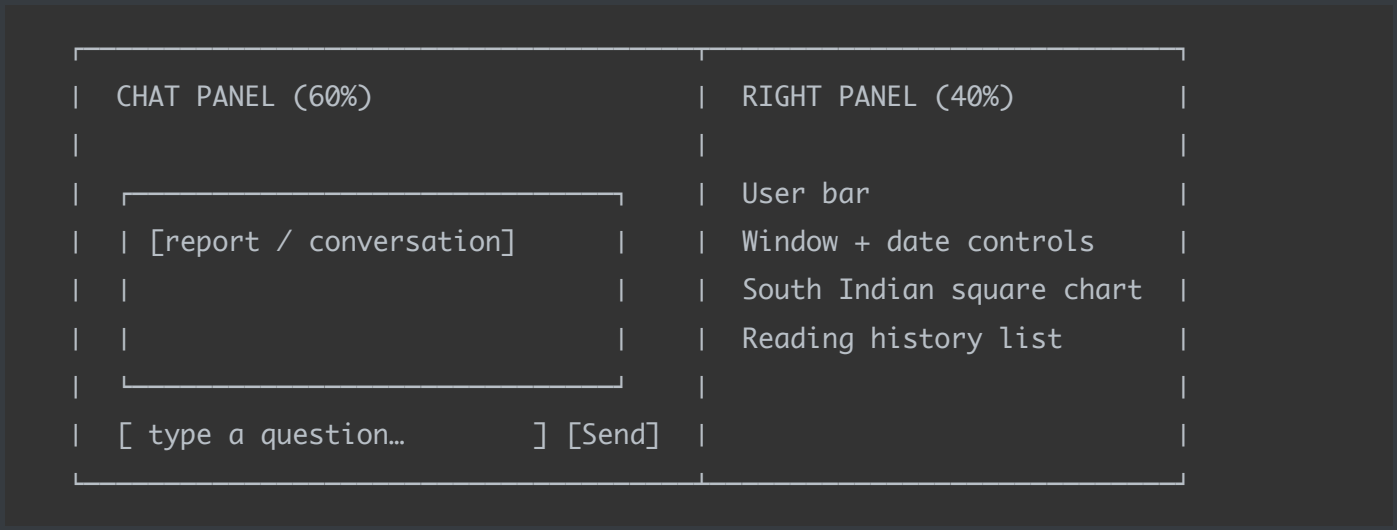
In v2, users can ask "show me my reading from last month" and the system fetches the full report from storage and injects it into the next Claude API call. The full_report field is stored in v1 in anticipation of this feature.

---

10. UI Design

10.1 Layout

Two-panel desktop layout. Mobile stacks vertically with the chart panel collapsible.



10.2 Chat Panel

- Displays Claude's transit report as formatted text on session start
- Subsequent turns rendered as a conversation: assistant bubbles and user bubbles
- Window selector (Day / Week / Month) + date picker in the header
- Send button disabled while Claude is streaming
- Streaming responses rendered token-by-token

10.3 Right Panel

User bar: name, birth date, lagna sign.

Window + date controls: pill buttons (Day / Week / Month), date picker. Selecting a new window or date triggers a new report.

Square chart: South Indian (Dakṣiṇa-śailī) format. Fixed sign positions, 4×4 grid with centre 2×2 empty. Planets listed by abbreviated IAST name in their sign cell. Lagna marked with "As" in the lagna cell. Transit overlay toggle: when active, transit planets shown in a distinct colour alongside natal planets.

South Indian sign grid (fixed positions):

```
  Pisces  | Aries  | Taurus | Gemini
  Aquarius|        |        | Cancer
  Capric. |        |        | Leo
  Sagitt. | Scorpio| Libra  | Virgo
```

Reading history list: date, window, top contact summary, period summary snippet. Tapping a history item opens the full report in the chat panel (v2).

10.4 Planet Abbreviations (UI)

Graha	Abbreviation
Sūrya	Su
Candra	Mo
Budha	Me
Śukra	Ve
Maṅgala	Ma
Guru	Ju
Śani	Sa
Rāhu	Ra
Ketu	Ke
Uranus	Ur
Neptune	Ne
Pluto	Pl
Lagna	As

Retrograde planets suffixed with ℞ in the chart cell.

10.5 Mobile

Chart panel collapses to a compact strip (sign + top 2 contacts listed as pills) above the chat. A "show chart" button expands it to full view. Reading history accessible via a bottom drawer.

---

11. Established Rules (Resolved)

11.1 Interpretation: LLM

All interpretation is generated by Claude. No rule-based templates, no pre-written interpretation strings. Claude receives the full symbolic framework via `symbol_map.json` and applies it natively.

11.2 Retrograde: Intensification + Inward Quality

Retrograde transit planets intensify the themes of that graha and add an inward, revisionary quality. Injected as explicit instruction in §6.4. Marked ℞ in output.

11.3 Bhāva = Whole-Sign

Bhāva assignment is always whole-sign. The sign a graha occupies = its house. No cusp calculation.

11.4 Moon Nakṣatra in Day Reports

Every day report includes a LUNAR WEATHER section. First-class section, not optional.

11.5 Transliteration: Always IAST

All Sanskrit terms use IAST in system prompts and all output. Examples: Śani, Candra, ātman, jñāna, māyā, Pūrvāṣāḍhā, Śatabhiṣaj.

11.6 Contact System: Parāśaran Dṛṣṭi + Yuti

Sign-based. No orbs. Yuti (co-presence) is distinct from dṛṣṭi (projection). Rāhu and Ketu cast special full dṛṣṭi to 5th and 9th per Parāśara.

11.7 User Model

v1: single-user, no auth, one profile per installation. v2: OAuth (Google), multi-user, practitioner model (Jyotiṣī manages multiple client profiles).

11.8 Chart Format

South Indian square (Dakṣiṇa-śailī). Fixed sign positions. This is the only chart format. No Western wheel chart.

11.9 Reading History

Lightweight summary (top contacts + period summary) injected into every session context. Full report stored in v1 for retrieval in v2. Default N = 5 most recent readings.

11.10 UI Theme

Two high-contrast themes, user-selectable and persisted in local storage: light (warm parchment, near-black ink, Cinzel + Spectral) and dark (deep navy, warm near-white, DM Serif Display + Merriweather + IBM Plex Mono). Both meet WCAG AA for all text. Theme toggle is accessible from the user bar in the sidebar.

11.12 LLM Provider: Claude-only (v1), Abstracted

v1 uses Anthropic Claude exclusively via `AnthropicClient`. All LLM calls go through the `LLMClient` abstract interface in `session/llm_client.py` — the session manager never imports the Anthropic SDK directly. v2 adds `OpenAIClient` and `GeminiClient` implementations. The onboarding form and settings page present only an Anthropic API key field in v1.

11.13 API Key Storage: OS Keychain

The Anthropic API key is stored in the OS keychain via `keyring` and never written to SQLite, any config file, or any file on disk. It is retrieved at FastAPI startup and held in memory as part of the shared `LLMClient` instance. The local server binds to `127.0.0.1` only. Claude outputs the initial transit report as newline-delimited JSON (NDJSON). Each line is one complete JSON object of a known type: `contact`, `period_summary`, `lunar_weather`, `history_note`. The server buffers the stream line by line, parses each object, and re-emits it as a typed SSE event. The client handles each event type directly, constructing the appropriate UI component from structured data — no prose parsing. Follow-up answers are plain prose. The `period_summary` is extracted from the `{"type":"period_summary",...}` object; no delimiters are needed.

---

12. Modules and File Structure

```
aspectarian/
├── engine/
|   ├── ephemeris.py          # swe setup, julday, ayanāṃśa, ephe path
|   ├── natal.py              # compute_chart() → NatalChart
|   ├── transit.py            # compute_transits() for date range
|   ├── contacts.py           # compute_contact(), score_contact() – dṛṣṭi +
yuti
|   ├── retrograde.py         # detect_retrograde()
|   ├── ketu.py               # derive Ketu from Rāhu
|   └── moon_nakshatra.py     # intra-day nakṣatra event detection
├── context/
```

```
|   ├── builder.py           # assemble_system_prompt() — pure string assembly
|   ├── history.py           # fetch + format reading history for injection
|   ├── symbol_loader.py     # load + validate symbol_map.json
|   └── iast.py              # IAST validation: detect Romanised Sanskrit in
output
├── session/
|   ├── manager.py           # conversation loop, API calls, session lifecycle
|   ├── llm_client.py        # LLMClient abstract base + AnthropicClient (v1)
|   ├── api_key.py           # keyring store/retrieve/delete for API key
|   ├── ndjson_parser.py     # streaming NDJSON line buffer + parser
|   └── extractor.py         # extract_period_summary(), extract_top_contacts()
├── storage/
|   ├── db.py                # SQLite connection, migrations
|   ├── users.py             # CRUD for User + NatalChart
|   ├── readings.py          # CRUD for Reading
|   └── sessions.py          # CRUD for Session
├── reports/
|   ├── day.py               # DayReport: single snapshot + lunar weather
|   ├── week.py              # WeekReport: 3-snapshot, Moon summary
|   └── month.py             # MonthReport: 4-snapshot, slow-planet focus
├── ui/
|   ├── app.py               # FastAPI entry point — serves static files + API
routes
|   └── static/              # HTML, CSS, JS (chart rendered client-side in JS)
├── api/                     # v2: REST API layer
|   ├── routes.py
|   ├── schemas.py
|   └── auth.py              # OAuth handlers (v2)
├── cli.py                   # v1 CLI entry point
├── config.py                # all tuneable parameters
└── symbol_map.json          # versioned interpretation framework
```

12.1 Key Interfaces

engine/contacts.py

```
    def compute_contact(transit_graha: str, transit_sign: int,
                        natal_graha: str, natal_sign: int) -> Contact | None
    def score_contact(contact: Contact) -> int
    def find_all_contacts(transits: TransitSnapshot, natal: NatalChart) ->
    list[Contact]
```

storage/db.py

```
    def init_db(path: str) -> sqlite3.Connection:
        """
        Open (or create) the SQLite database and ensure all tables exist.
        Uses CREATE TABLE IF NOT EXISTS for every table — no migration framework.
        Schema changes in future versions add columns via ALTER TABLE IF NOT EXISTS
        (SQLite 3.37+), written as explicit upgrade functions called after init.
        GeoNames import runs here on first launch if the cities table is empty.
        """
```

session/llm_client.py

```
    from abc import ABC, abstractmethod
    from typing import AsyncGenerator

    class LLMClient(ABC):
        """
        Abstract streaming LLM interface. v1 has one implementation:
    AnthropicClient.
        v2 will add OpenAIClient, GeminiClient. The session manager depends only on
        this interface — never on the Anthropic SDK directly.
        """
        @abstractmethod
        async def stream(
            self,
            system:     str,
            messages:   list[dict],
            max_tokens: int,
        ) -> AsyncGenerator[str, None]:
            """Yield raw text tokens as they stream from the model."""
            ...
```

```python
class AnthropicClient(LLMClient):
    def __init__(self, api_key: str, model: str):
        import anthropic
        self._client = anthropic.Anthropic(api_key=api_key)
        self._model  = model

    async def stream(self, system, messages, max_tokens):
        async with self._client.messages.stream(
            model      = self._model,
            max_tokens = max_tokens,
            system     = system,
            messages   = messages,
        ) as s:
            async for token in s.text_stream:
                yield token

    @classmethod
    def validate_key(cls, api_key: str) -> bool:
        """
        Validate an API key by making the cheapest possible test call.
        Uses claude-haiku (max_tokens=1) - costs < $0.001.
        Returns True if auth succeeds, False on AuthenticationError.
        """
        import anthropic
        try:
            anthropic.Anthropic(api_key=api_key).messages.create(
                model      = "claude-haiku-4-5-20251001",
                max_tokens = 1,
                messages   = [{"role": "user", "content": "hi"}],
            )
            return True
        except anthropic.AuthenticationError:
            return False


def get_llm_client() -> LLMClient:
    """
    Instantiate the configured LLM client using the stored API key.
    Called once at FastAPI startup; the instance is shared across all requests.
```

```
    Raises RuntimeError if no API key is stored (user has not completed
onboarding).
    """
    from session.api_key import get_api_key
    key = get_api_key()
    if not key:
        raise RuntimeError("No API key found. Complete onboarding first.")
    return AnthropicClient(api_key=key, model=settings.CLAUDE_MODEL)
```

session/api_key.py

```
import keyring

_SERVICE = "aspectarian"
_ACCOUNT = "anthropic_api_key"

def store_api_key(key: str) -> None:
    """Store API key in the OS keychain (macOS Keychain, Windows Credential
    Manager, Linux Secret Service). Never written to disk or SQLite."""
    keyring.set_password(_SERVICE, _ACCOUNT, key)

def get_api_key() -> str | None:
    """Retrieve API key from the OS keychain. Returns None if not set."""
    return keyring.get_password(_SERVICE, _ACCOUNT)

def delete_api_key() -> None:
    """Remove API key from keychain. Called if user resets the app."""
    try:
        keyring.delete_password(_SERVICE, _ACCOUNT)
    except keyring.errors.PasswordDeleteError:
        pass   # already absent
```

context/symbol_loader.py

```
# Module-level cache — loaded once on first import, reused for all sessions.
_symbol_map: dict | None = None

def load_symbol_map(path: str = settings.SYMBOL_MAP_PATH) -> dict:
    """
```

```
    Load and validate symbol_map.json. Cached in module scope after first call.
    Raises RuntimeError on startup if any required top-level key is missing.
    """
    global _symbol_map
    if _symbol_map is not None:
        return _symbol_map

    with open(path, encoding='utf-8') as f:
        data = json.load(f)

    REQUIRED_KEYS = {'graha', 'rasi', 'bhava', 'contacts',
                     'guna_system', 'correlations', 'interpretation_rules'}
    missing = REQUIRED_KEYS - data.keys()
    if missing:
        raise RuntimeError(f"symbol_map.json missing required keys: {missing}")

    _symbol_map = data
    return _symbol_map
```

Called once during FastAPI startup via a `lifespan` handler; subsequent calls return the cached dict instantly.

`context/iast.py` that should never appear in Claude output. ROMANISED = ["Shani", "Chandra", "Brahma", "Vishnu", "Shiva", "Rahu", "Ketu", "nakshatra", "drishti", "graha", "rashi", "bhava", "guna"]

def validate_iast(text: str) -> list[str]: """Return list of Romanised terms found in text. Empty list = clean."""" return [w for w in ROMANISED if w in text]

```
Used in the IAST validation test and optionally as a post-processing assertion
on Claude output during development.

**`context/builder.py`**
```python
def assemble_system_prompt(
    natal:        NatalChart,
    transits:     TransitSnapshot,
    contacts:     list[Contact],
    symbol_map:   dict,
    history:      list[Reading],
```
```

```
    window:       str,
    ref_start:    date,
    ref_end:      date,
    moon_events:  list[NakshatraEvent] | None = None,
) -> str
# Full implementation in §6.14
```

storage/readings.py

```
def save_reading(reading: Reading) -> str           # returns reading_id
def get_history(user_id: str, n: int) -> list[Reading]
def get_reading(reading_id: str) -> Reading
```

session/extractor.py

```
def extract_period_summary(report_objects: list[dict]) -> str:
    """
    Extract the period summary from the parsed NDJSON report objects.
    Claude outputs one {"type":"period_summary","text":"..."} object per report.
    Raises ValueError if absent — session is still saved, period_summary stored
    as empty string, reading flagged for review.
    """
    for obj in report_objects:
        if obj.get('type') == 'period_summary':
            return obj.get('text', '')
    raise ValueError("No period_summary object in report")

def extract_top_contacts(contacts: list[Contact], n: int = 3) ->
list[ContactSummary]:
    """Return the top n contacts by score, for reading history injection."""
    return sorted(contacts, key=lambda c: c.score, reverse=True)[:n]
```

session/ndjson_parser.py (new module)

```
import json

def parse_stream(stream_text: str) -> tuple[list[dict], str]:
    """
    Parse complete NDJSON lines from a stream buffer.
```

```
    Returns (parsed_objects, remaining_buffer).
    Accumulates incomplete lines back into the buffer.
    On JSONDecodeError for a complete line, logs the error and skips the line.
    """
    objects = []
    lines = stream_text.split('\n')
    remaining = lines.pop()      # last element may be incomplete
    for line in lines:
        line = line.strip()
        if not line:
            continue
        try:
            objects.append(json.loads(line))
        except json.JSONDecodeError as e:
            logger.warning(f"NDJSON parse error: {e} — line: {line[:80]}")
    return objects, remaining
```

`context/iast.py`

```
# Known Romanised forms that should never appear in Claude output.
ROMANISED = ["Shani", "Chandra", "Brahma", "Vishnu", "Shiva", "Rahu", "Ketu",
             "nakshatra", "drishti", "graha", "rashi", "bhava", "guna"]

def validate_iast(text: str) -> list[str]:
    """Return list of Romanised terms found in text. Empty list = clean."""
    return [w for w in ROMANISED if w in text]
```

Used in the IAST validation test and optionally as a post-processing assertion on Claude output during development.

13. Configuration (`config.py`)

Pattern: Pydantic `BaseSettings` — reads from environment variables and a `.env` file in the project root. All parameters have defaults; none are required for first run.

Parameter	Default	Description
`EPHE_PATH`	`./ephemeris/ephe`	Swiss Ephemeris data directory
`AYANAMSA`	`SIDM_LAHIRI`	Ayanāṃśa mode
`HOUSE_SYSTEM`	`W`	Whole-sign (swe flag)
`DRSTI_MIN_GRADE`	`"quarter"`	Server-side default; overridden per user by `UserPreferences`
`INCLUDE_KETU`	`True`	Derive and include Ketu
`INCLUDE_OUTER_PLANETS`	`True`	Uranus, Neptune, Pluto
`TRANSIT_SAMPLE_HOUR_UTC`	`6`	Hour for transit snapshot (sunrise proxy)
`SYMBOL_MAP_PATH`	`./symbol_map.json`	Interpretation framework file
`CLAUDE_MODEL`	`claude-sonnet-4-20250514`	Model passed to `AnthropicClient`
`CLAUDE_MAX_TOKENS`	`4096`	Max tokens per Claude response
`HOST`	`127.0.0.1`	Server bind address — localhost only, never `0.0.0.0`
`NATAL_CACHE_TTL`	`∞`	Natal charts cached permanently
`TRANSIT_CACHE_TTL`	`3600`	Transit positions cached for 1 hour (seconds)
`DB_PATH`	`./aspectarian.db`	SQLite database file path
`GEONAMES_TXT`	`./data/cities15000.txt`	GeoNames city data source
`GEONAMES_DB`	`./data/geonames.db`	GeoNames SQLite database (built on first run)
`HISTORY_MAX_READINGS`	`5`	Max readings injected into context
`PORT`	`7842`	Local server port
`OUTPUT_FORMAT`	`text`	CLI: `text` or `json`

---

14. Report Windows

Day Report

- Transit snapshot at `TRANSIT_SAMPLE_HOUR_UTC` on the target date
- All contacts (yuti + all dṛṣṭi grades ≥ `DRSTI_MIN_GRADE`) included
- Lunar weather section mandatory
- Report saved to Reading on session end
- Expected output: 3–8 contacts + lunar weather + period summary

Week Report

- Transit snapshots at start, mid, and end of the 7-day window
- Contact included if active in ≥ 2 of 3 snapshots
- Moon's nakṣatra changes summarised as a day-by-day list
- Expected output: 5–12 contacts + Moon summary + period summary

Month Report

- Transit snapshots weekly (4 snapshots)
- Only slow-planet contacts included individually
- Fast-planet contacts synthesised into one thematic paragraph
- Retrograde ingresses and stations flagged if within the month
- Expected output: 4–10 slow-planet contacts + fast-planet synthesis + period summary

---

15. CLI Interface (v1)

```
# First run — enter API key, create user profile
python -m aspectarian.cli setup

# Generate and run a weekly session
python -m aspectarian.cli session --window week

# Generate a day report for a specific date, non-interactive
python -m aspectarian.cli report --window day --date 2026-04-12 --output-format
json

# List past readings
python -m aspectarian.cli history --n 10
```

`cli setup` flow:

1. Prompt for Anthropic API key → call `AnthropicClient.validate_key()` → store via `store_api_key()`
2. Prompt for name
3. Prompt for city (fuzzy search against GeoNames) → fills lat/lon/timezone
4. Prompt for birth date and time (local) → convert to UTC via `local_to_utc()`
5. Prompt for preferences (theme, default window, grade filter, display options)
6. Write profile to SQLite

In session mode: initial report is printed progressively (object by object), then the CLI drops into a REPL for follow-up questions. On `exit` or `quit`, the session is closed.

---

16. REST API (v2)

Base URL: `/api/v1/`
Auth: None in v1 (single-user). Bearer token via OAuth in v2.
Content-Type: `application/json` for all non-SSE responses.

Pydantic Schemas

```
# ── Request bodies ──────────────────────────────────────────

class CreateUserRequest(BaseModel):
    name: str                       # non-empty
    birth_datetime_utc: datetime    # past datetime
    latitude: float                 # -90.0 to 90.0
    longitude: float                # -180.0 to 180.0
    place_name: str | None = None
    timezone: str | None = None     # IANA tz string, display only

class UpdateUserRequest(BaseModel):
    name: str | None = None
    place_name: str | None = None
    timezone: str | None = None
    # Birth data fields — changing any of these invalidates the natal chart
cache
```

```python
    birth_datetime_utc: datetime | None = None
    latitude: float | None = None
    longitude: float | None = None

class StartSessionRequest(BaseModel):
    window: Literal['day', 'week', 'month']
    ref_date: date                  # YYYY-MM-DD; defaults to today if omitted

class SendMessageRequest(BaseModel):
    message: str = Field(min_length=1, max_length=2000)

# ── Response bodies ──────────────────────────────────────

class UserResponse(BaseModel):
    id: str
    name: str
    birth_datetime_utc: str         # ISO 8601
    latitude: float
    longitude: float
    place_name: str | None
    timezone: str | None
    created_at: str

class ContactSummaryResponse(BaseModel):
    transit: str
    natal: str
    contact_type: str               # 'yuti' | 'drsti'
    drsti_grade: str | None
    strength: str

class ReadingSummaryResponse(BaseModel):
    id: str
    window: str
    ref_start: str
    ref_end: str
    top_contacts: list[ContactSummaryResponse]
    period_summary: str
    created_at: str
    # full_report NOT included in list view
```

```python
class ReadingDetailResponse(ReadingSummaryResponse):
    full_report: str                # included only on GET /readings/{id}

class ReadingsListResponse(BaseModel):
    readings: list[ReadingSummaryResponse]
    total: int
    offset: int
    limit: int

class SessionResponse(BaseModel):
    id: str
    user_id: str
    reading_id: str | None
    conversation_history: list[dict]
    created_at: str
    last_active: str
```

Error Handling

All endpoints return a consistent error body:

```
{ "error": "string", "detail": "string (optional)" }
```

Status	When
`400`	Malformed request body
`404`	User, reading, or session not found
`409`	Birth data update produces invalid coordinates or datetime
`422`	Pydantic validation failure (FastAPI default)
`500`	Swiss Ephemeris failure, SQLite write failure, Claude API failure

SSE routes yield a final typed error event before closing on failure:

```
event: error
data: {"message": "Claude API error: 529 overloaded"}
```

`POST /api/validate-key`

Validates an Anthropic API key without storing it. Called from the onboarding wizard and the settings page before storing a new key. Uses `AnthropicClient.validate_key()` internally.

```
Request body: { "api_key": "sk-ant-..." }
200 OK → { "valid": true }
200 OK → { "valid": false, "error": "Invalid API key" }
```

Note: always returns 200 — validation failure is a business result, not an HTTP error.

`POST /api/setup/api-key`

Validates and stores the API key in the OS keychain. Called once during onboarding after successful validation.

```
Request body: { "api_key": "sk-ant-..." }
200 OK → { "stored": true }
400 Bad Request → { "error": "API key validation failed" }
```

`PUT /api/users/me/api-key`

Replaces a stored API key. Called from the settings page. Validates before storing.

```
Request body: { "api_key": "sk-ant-..." }
200 OK → { "stored": true }
400 Bad Request → { "error": "API key validation failed" }
```

Security: The server binds to `127.0.0.1` only (not `0.0.0.0`). The API key travels only over localhost — it never leaves the machine. The key is stored in the OS keychain and never written to SQLite or any file on disk.

`GET /users/me`

Returns the single user profile. 404 if no profile has been created (triggers client redirect to `/setup`).

```
200 OK → UserResponse
404 Not Found → { "error": "No user profile found" }
```

`POST /users`

Creates the user profile on first run. Only one profile exists in v1 — returns 409 if one already exists.

```
Request body: CreateUserRequest
201 Created → UserResponse
409 Conflict → { "error": "User profile already exists" }
```

`PUT /api/users/me/preferences`

Updates one or more preference fields. Accepts partial updates — only send the keys that changed. Returns the full updated `UserPreferences` object.

```
Request body: partial UserPreferences (all fields optional)
200 OK → UserPreferences
```

Called from both the settings page (on each pill click) and the `ThemeToggle` in the main app header.

Updates profile fields. If any birth data field changes, the natal chart cache is invalidated (the `natal_charts` row is deleted; chart is recomputed on next `GET /users/me/chart`).

```
Request body: UpdateUserRequest  (all fields optional)
200 OK → UserResponse
409 Conflict → { "error": "Invalid birth data", "detail": "..." }
```

Cache invalidation logic:

```
BIRTH_DATA_FIELDS = {'birth_datetime_utc', 'latitude', 'longitude'}
if any(f in request.model_fields_set for f in BIRTH_DATA_FIELDS):
    await db.delete_natal_chart(user.id)
```

`GET /users/me/chart`

Returns the cached natal chart. On cache miss, computes it synchronously from birth data (typically <500ms), stores it, and returns it.

```
200 OK → NatalChart JSON (the full chart_data blob)
```

Cache logic:

```
chart = await db.get_natal_chart(user.id)
if not chart:
    chart = compute_chart(user.birth_datetime_utc, user.latitude,
user.longitude)
    await db.save_natal_chart(user.id, chart)
return chart
```

`GET /users/me/readings`

Returns paginated reading history, sorted by `created_at` descending.

```
Query params:
  limit  int  default=10, max=50
  offset int  default=0

200 OK → ReadingsListResponse
```

Example response:

```
{
  "readings": [ { ...ReadingSummaryResponse... } ],
  "total": 12,
  "offset": 0,
  "limit": 10
}
```

`GET /users/me/readings/{id}`

Returns a single reading including the full Claude report text.

```
200 OK → ReadingDetailResponse
404 Not Found → { "error": "Reading not found" }
```

`POST /sessions`

Starts a new session and streams the initial NDJSON transit report. Fully specced in §7.2 (session flow) and §9.1 (SSE implementation).

```
Request body: StartSessionRequest
Response: SSE stream
  event: contact        data: <Contact NDJSON object>
  event: period_summary data: <PeriodSummary NDJSON object>
  event: lunar_weather  data: <LunarWeather NDJSON object>  (day only)
  event: history_note   data: <HistoryNote NDJSON object>   (if history present)
  event: done           data: {"session_id": "...", "reading_id": "..."}
  event: error          data: {"message": "..."}            (on failure)
```

The reading is saved before `event: done` is emitted. The client does not need to call DELETE to trigger saving.

`POST /sessions/{id}/messages`

Sends a follow-up message and streams Claude's plain prose response. The system prompt is unchanged; the user message is appended to the conversation history before the API call.

```
Request body: SendMessageRequest
404 Not Found → { "error": "Session not found" }
Response: SSE stream (plain prose tokens — NOT NDJSON)
  event: token  data: <plain text chunk>
  event: done   data: {"session_id": "...", "tokens": <int>}
  event: error  data: {"message": "..."}
```

Server implementation:

```python
@app.post("/api/sessions/{session_id}/messages")
async def send_message_route(session_id: str, body: SendMessageRequest):
    session = await db.get_session(session_id)
```

```
    if not session:
        raise HTTPException(404)

    async def generate():
        async for token in session_manager.send_message(
            session_id = session_id,
            message    = body.message,
            llm_client = app.state.llm_client,   # injected at startup
        ):
            yield {"event": "token", "data": token}

        yield {"event": "done", "data": json.dumps({"session_id": session_id})}

    return EventSourceResponse(generate())
```

`app.state.llm_client` is the shared `LLMClient` instance created at FastAPI startup via `get_llm_client()`. The session manager never imports the Anthropic SDK directly.

`GET /sessions/{id}`

Returns session state and full conversation history. Primarily used by the UI to restore state after a page refresh.

```
200 OK → SessionResponse
404 Not Found → { "error": "Session not found" }
```

`DELETE /sessions/{id}`

Marks the session inactive. In v1 the reading is already saved by the time `POST /sessions` SSE completes, so this endpoint performs no extraction — it only updates `last_active` and flags the session as closed.

```
200 OK → { "session_id": "...", "status": "closed" }
404 Not Found → { "error": "Session not found" }
```

---

17. `symbol_map.json` — Status

The file is complete as of v1.2. It covers all 12 grahas (including Ketu with dual-natured guṇa, and Uranus/Neptune/Pluto as modern outer planets), all 12 rāśis (Parāśaran elemental guṇa assignments), all 12 bhāvas, the full Parāśaran contact system (yuti + all dṛṣṭi grades + special dṛṣṭi for Maṅgala/Guru/Śani/Rāhu/Ketu), and consciousness/doṣa correlations for all 12 grahas.

Validation on startup: `symbol_loader.py` checks for all required top-level keys on load and raises `RuntimeError` if any are missing. Add new graha/rāśi/bhāva entries to the appropriate sections without changing the top-level structure.

---

18. Testing Plan

Test	Method
Planetary position accuracy	Compare against Jagannatha Hora / Astro-Seek for known dates
Ketu derivation	Assert `ketu_lon == (rahu_lon + 180) % 360`
Retrograde detection	Verify against known retrograde periods
Whole-sign bhāva assignment	Unit test lagna → bhāva mapping for all 12 signs
House count calculation	Unit test `compute_contact()` for all 12 house count values across all sign pairs
Yuti detection	Assert `contact_type == "yuti"` when transit and natal in same sign
Base dṛṣṭi grades	Assert correct grade for houses 3, 4, 5, 7, 8, 9, 10 for non-special planet
No-contact houses	Assert `None` for house counts 2, 6, 11, 12
Special dṛṣṭi upgrade	Assert Maṅgala 4/8, Guru 5/9, Śani 3/10, Rāhu 5/9 → `grade="full"`, `is_special=True`
DRSTI_MIN_GRADE filter	Assert quarter contacts excluded when `DRSTI_MIN_GRADE = "half"`
System prompt completeness	Assert all required XML blocks present in assembled prompt
System prompt block order	Assert blocks appear in specified order (role → framework → natal → transits → contacts → ...)
Retrograde block conditional	Assert `<retrograde_planets>` present when retrogrades exist; absent when none

Reading history conditional	Assert `<reading_history>` present when history exists; absent when empty
Moon events conditional	Assert `<moon_nakshatra_events>` present for day reports; absent for week/month
Contact input dict	Assert `Contact.to_context_dict()` excludes headline, body, guna, consciousness, tone
Reading summary dict	Assert `Reading.to_summary_dict()` excludes full_report
wrap_xml	Assert output is `<tag>\ncontent\n</tag>`
History injection	Assert reading history block present and ordered oldest-first
History omission	Assert block is omitted when no readings exist
NDJSON parse — happy path	Assert `parse_stream()` correctly parses all four object types from a well-formed NDJSON string
NDJSON parse — incomplete line	Assert incomplete final line is returned as `remaining_buffer`, not parsed
NDJSON parse — malformed line	Assert malformed line is logged and skipped; remaining objects still returned
Extractor — period summary	Assert `extract_period_summary()` returns `text` field from `period_summary` object; raises `ValueError` if absent
Extractor — top contacts	Assert top 3 contacts returned in score order
Reading save/retrieve	Write a Reading; fetch history; assert field integrity
API — GET /users/me	200 with profile; 404 before setup
API — POST /users	201 creates profile; 409 on duplicate
API — PUT /users/me birth data	Updates fields; asserts natal_charts row deleted
API — PUT /users/me safe fields	Updates name/place; asserts natal_charts row intact
API — GET /users/me/chart cache hit	Returns stored chart without calling compute_chart()
API — GET /users/me/chart cache miss	Calls compute_chart(), stores result, returns chart
API — GET	Returns correct slice; total matches DB count

/users/me/readings pagination	
API — GET /users/me/readings/{id}	Returns full_report field; 404 on unknown id
API — POST /sessions/{id}/messages	Appends user + assistant turns to conversation_history
API — DELETE /sessions/{id}	Sets status to 'closed'; 404 on unknown id
API key — validate good key	Assert `AnthropicClient.validate_key()` returns True for a valid key
API key — validate bad key	Assert returns False (not raises) for an invalid key
API key — keyring round-trip	`store_api_key(k)` → `get_api_key()` returns k → `delete_api_key()` → `get_api_key()` returns None
API key — missing key at startup	Assert `get_llm_client()` raises RuntimeError when no key stored
API — POST /api/validate-key valid	Returns `{"valid": true}`
API — POST /api/validate-key invalid	Returns `{"valid": false}` with 200 status
API — POST /api/setup/api-key	Stores key in keychain; subsequent `get_api_key()` returns it
Session persistence	Assert conversation_history grows correctly across turns
Chart renderer	Assert SVG output contains correct sign labels and planet placements
IAST validation	Assert no Romanised Sanskrit appears in output
`symbol_map.json` schema	Validate on load; fail loudly on missing keys

---

19. Versioning

Version	Milestone
v1.0	CLI, computation engine (Ketu, retrograde, dṛṣṭi + yuti), context builder, Claude API, day/week/month reports, session Q&A, SQLite persistence, single-user profile, reading history, GeoNames location, keyring API key
v1.1	Web UI — South Indian chart viewer, chat interface, reading history panel, onboarding wizard, settings page
v2.0	OAuth (Google), multi-user, PostgreSQL, REST API, streaming Claude responses, topocentric Moon correction
v2.1	Practitioner model (Jyotiṣī manages multiple client profiles); Viṃśottarī daśā awareness (`engine/dasa.py` , `<dasa_period>` context block)
v2.2	Daśā report window (4th window type alongside day / week / month); full reading history retrieval in session context
v2.3	Daśā-transit synthesis (daśā lord score modifier in `score_contact()`); pratyantardaśā-level timing in reports
v3.0	Mobile app (React Native); PDF report export

End of Specification — Aspectarian v1.4 Draft