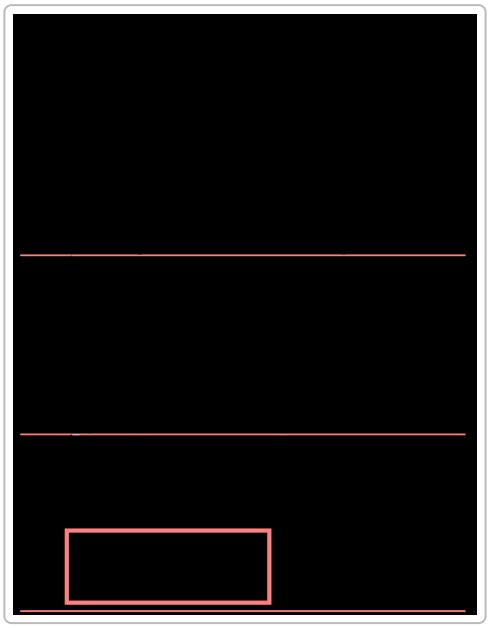
ChatGPT

Designing a Multi-Language Biblical Gematria Module

Greek Gematria in Initial Scope: Include or Defer?

Including **Greek isopsephy (gematria)** in the first release depends on project priorities. The core module plan already anticipates Greek letter-to-number mapping "if needed" [1]. If the immediate focus is Hebrew Scripture analysis, it's reasonable to defer Greek gematria until the Hebrew features stabilize. This keeps the initial scope lean. However, if the system aims to analyze New Testament numerics or cross-language patterns early on, incorporating Greek gematria from the start ensures completeness. In practice, adding Greek support is not overly complex (the mapping is well-known), so designing the module to accommodate it (even if turned off until needed) is wise. In summary, **defer Greek gematria for v1 unless NT analysis is a key use-case**, but architect the module so Greek can be added easily later without breaking anything.

Greek Isopsephy Letter-to-Number Mapping and Variants



Basic Hebrew and Greek gematria chart. Greek **isopsephy** assigns values to letters using the classical 27-letter scheme: the 24 Greek letters plus 3 obsolete characters to fill gaps [2]. In this system, alpha = 1, beta = 2, ..., iota = 10, ... phi = 500, chi = 600, psi = 700, omega = 800. To cover the numbers 6, 90, and 900, Greek uses the archaic letters **digamma** (Ϝ, later written as stigma ϛ) = 6, **qoppa** (Ϙ) = 90, and **sampi** (ϡ) = 900 [2]. Each letter's value is added to get a word's total [3]. This mapping should be sourced from authoritative references to avoid mistakes – for example, *Wikipedia* confirms the 1–9, 10–90, 100–900 assignments including digamma/stigma, qoppa, and sampi [2]. In modern Greek usage (which no longer uses

digamma), the combination “στ” is sometimes used to represent 6 [4], but the underlying values remain the same. Alternate or “extended” mappings beyond the classical system are generally not used – the standard ancient scheme is the de facto mapping for isopsephy. Therefore, the implementation should use the **classical values** (with an option to recognize stigma ς or “στ” as 6 if needed) and need not worry about modern variants. Thorough unit tests using known examples (e.g. Νέρων = 1005 [5] or the infamous 666 in Revelation) can verify the correctness of the Greek mapping.

Unicode Normalization for Greek Text

Handling Greek text input requires careful **Unicode normalization** so that different representations of the same word yield the same gematria result. Biblical Greek words often include polytonic diacritics (accent and breathing marks) and may use different codepoint compositions. The module should normalize these by: (1) converting to a consistent case (e.g. all lowercase), (2) decomposing combined characters and stripping diacritics, and (3) reconciling final sigma. A recommended approach is **NFD normalization** (or NFKD for compatibility), removal of combining marks, then recomposition (NFC) [6]. For example, *ἀντίθεσις* (with polytonic accent) and *αντίθεσις* (modern monotonic) should both normalize to “αντιθεσις” (accents removed, final ς handled) [7]. Ensure that the final sigma (U+03C2 “ς”) is replaced with the standard sigma (U+03C3 “σ”) during normalization [8], since Unicode won’t do this by default. By stripping **all diacritical marks** (acute, grave, circumflex, smooth/rough breathing, diaeresis) which are represented as combining characters [9] [10], and mapping final sigma to sigma, the Greek text is reduced to its bare letters for gematria calculation. This process is analogous to the Hebrew normalization pipeline (NFKD + strip cantillation/points + NFC) already defined in the project [6]. In summary, implement a Unicode normalization util for Greek that removes accents and breathings and normalizes sigma, to ensure consistent, accurate numeric calculations.

Supporting Multiple Hebrew Gematria Systems (Mispar Gadol, Shemi, Katan)

It is best practice for a gematria module to support **multiple Hebrew gematria methods**, since different traditions use different calculations. The standard system (Mispar *Hechrachi*, a.k.a. absolute or normative value) is most common [11] and should be the default. In Mispar Hechrachi, each of the 22 Hebrew letters has its conventional value (א=1, ב=2, … י=10, כ=20, … ת=400) [11], and usually the five final letters share the same values as their non-final forms [12]. The module can first implement this normative scheme.

For completeness, the module can also include **Mispar Gadol** (“large number” method) where the five final letters are given high values (ך=500, ם=600, ן=700, ף=800, ץ=900) [13]. This extended scheme treats the final forms as continuation of the alphabet and allows numeric values up to 900 for single letters. It has precedent in Kabbalistic literature – as noted by one source, sometimes final *kaf* is counted as 500, *final mem* as 600, etc., completing a cycle where ת=400 and final ץ=900 [14]. The **Mispar Katan** (reduced value) method is another variant: it strips the zeros from the standard values, effectively using each letter’s value mod 10 (e.g. י which is 10 becomes 1, כ/20 becomes 2, ת/400 becomes 4) [15]. This yields single-digit values 1–9 for letters and is used in some numerological analyses. A more complex method is **Mispar Shemi** (a.k.a. Milui, full name), which calculates the value of each letter by summing the values of the letters in that letter’s name [16]. For example, *Aleph* (אלף) is 1+30+80 = 111 [16]. Mispar Shemi requires a predefined spelling for each letter name – note that some letters have variant spellings, so one must choose a standard (the module should document which spelling convention it uses to ensure reproducibility [16]).

In terms of **prioritization**, the module should treat the **Standard (Mispar Hechrachi)** as the default/fallback for general use [11], since this is the normative method in most Hebrew studies and commentaries. Mispar Gadol and Katan can be offered as optional calculations for users specifically looking for those patterns. (For instance, some gematria analyses look for matches only if including final-letter values.) Mispar Shemi is more esoteric; it could be implemented for advanced users or for completeness, but it may be reasonable to defer it initially due to its complexity and less frequent use. The key is to **decouple the logic** so that adding a new calculation method is easy – e.g. by having a parameter or separate function for each method. Also, document any assumptions (like whether final letter values are used by default – traditional sources indicate they "generally are given the same value" as normal letters, with the 500–900 usage being an alternative [12]). By supporting multiple systems in one module (via configurable modes or functions), the system can cater to various interpretive traditions while keeping all gematria logic centralized.

Handling Ketiv/Qere Variant Readings

Ketiv/Qere (written vs. read variants in the Hebrew Bible) must be handled consistently in both numeric calculations and AI-driven noun discovery. The Ketiv (כתיב, "written" form) represents the consonants as preserved in the text, whereas the Qere (קרי, "read" form) is the variant reading that the Masoretes indicate should be read aloud. For gematria calculations, it is generally advisable to use the *Ketiv* – i.e. the actual letters written in the canonical text – because gematria is inherently tied to the letters as written. Using the written form ensures we are calculating the intended textual value, even if the pronounced word differs. Many traditional commentaries that employ gematria do in fact pay attention to the *written* letters (sometimes finding significance in anomalous spellings, which would be lost if one used the Qere). Therefore, the module should calculate numeric values based on the Ketiv by default, while perhaps noting the Qere in metadata. This approach aligns with the design rule that **Ketiv is primary and variants are recorded** [17] – the noun discovery feature already follows this rule by treating the Ketiv spelling as the main entry and logging the Qere as a variant.

To maintain consistency, the **AI-noun discovery** module should similarly treat the Ketiv form of a word as the key form for indexing and analysis, but should be aware of Qere for completeness (e.g. it might tag a discovered noun as having a Qere variant). If a Qere is significantly different (different word entirely), the system could optionally compute its gematria as well, but it should not mix the two in one calculation. The important part is **consistency**: both gematria and noun-extraction pipelines should follow the same policy so that, for example, a word isn't counted twice or missed due to one module using Ketiv and another using Qere. By using the Ketiv uniformly, we preserve a single source of truth for the text's letters, which is doctrinally sound (the written consonantal text is the basis of the Masoretic tradition) and programmatically simpler. The Qere can be included in outputs or logs for reference, but not as the primary basis for numeric or lexical computations.

Explaining Hebrew/Greek Morphology Codes in the Lexicon Adapter

The lexicon adapter needs to provide human-readable explanations of morphology/grammar codes (e.g. parsing codes like "Vqp3ms" or "N-AMS" etc.) for Hebrew and Greek words. The plan identifies this as a future enhancement [18] [19], and importantly, it indicates that the data for these explanations is stored in the database. Indeed, the **bible_db** schema includes tables such as `bible.hebrew_morphology_codes`

and `bible.greek_morphology_codes` with fields for the code and its description (part of speech, tense, voice, etc.) [20]. The best practice is to leverage these tables rather than trying to generate explanations on the fly. This ensures **consistency and accuracy** (the codes have standard meanings defined by scholarly sources or the database provider).

To implement this, the lexicon adapter (or the lexicon flow) can perform a **join query** or a lookup when retrieving a lexicon entry or word. For example, when a user looks up a word by Strong's number, the system can fetch the word's morphology code (from `hebrew_ot_words` or `greek_nt_words`) and then look up the code in the corresponding morphology_codes table to get the full text explanation [21]. The adapter can then return or display something like "**Verb – Qal perfect 3ms**" instead of just "vq3ms". This should be done in a **DB-first** manner – i.e. the explanations come straight from the DB tables, not from an LLM. Since the tables are read-only reference data, this fits perfectly with the system's philosophy of using the database as the source of truth for biblical data.

In terms of design: one could load the entire morphology code table into memory as a dictionary at startup (they are usually reasonably small, a few hundred entries) for quick lookup, or query on the fly with a cached prepared statement. Either approach maintains the read-only, data-driven pattern. By fulfilling this enhancement, the system will significantly improve usability – users will see grammar codes explained in plain language (for both Hebrew and Greek), facilitating better understanding of lexicon results. This also aligns with the "future enhancement" note in the plan [18], ensuring the lexicon adapter provides not just raw data but also interpretive help.

Cross-Language Flow: Retrieving Actual Hebrew Lemma from Lexicon

The `cross_language_flow.py` is meant to enable cross-language word analysis – e.g. linking a Greek word to a Hebrew word or vice versa. The TODO "Get actual Hebrew lemma from lexicon table" likely refers to retrieving the original Hebrew word (in Hebrew characters) for a given Strong's number, rather than just using an English gloss or a transliteration. To achieve this while respecting the system's **DB-first, read-only** approach, the flow should query the **lexicon tables** for the lemma. For example, if analyzing a Hebrew Strong's #, one can look up the `bible.hebrew_entries` table to get the `lemma` field (which contains the Hebrew spelling of the lemma) [22]. Similarly, for a Greek Strong's number, `bible.greek_entries.lemma` gives the Greek lemma in Greek script [23]. This is preferable to relying on the `word_text` from the words table because the words table might have the inflected form or may not include the full lemma in some cases, and certainly better than using an LLM to guess the lemma.

By **joining or querying the lexicon entry** directly, you ensure you get the exact authorized lemma from the database. This maintains consistency (the same lemma used in lexicon definitions will be fetched) and keeps with the read-only pattern (just reading another table). It appears the cross-language flow was already designed to use the lexicon adapter and vector search [24]; the missing piece was likely just pulling the actual Hebrew text. Implement this by calling the lexicon adapter's lookup (e.g. `lexicon_adapter.get_lemma(strongs_id, language)`) which returns the lemma string. Internally, that adapter can simply do a SELECT on the appropriate table by strongs_id. This way, the cross-language feature can display "Hebrew Lemma: בראשית (bereshit)" instead of only an English gloss or transliteration for a Hebrew word, for instance. This small addition will make cross-language comparisons clearer (since

users can see the actual original word side by side with its counterpart). And because it's done via the DB and existing adapter, it doesn't violate the **DB-first** or **read-only** constraints.

In summary, **fetch lemmas from the DB lexicon tables** rather than from any AI or manual mapping. The design remains clean: the flow triggers a DB lookup through the adapter, gets the lemma (Hebrew or Greek script), and uses it for display or further search (like finding where else that lemma occurs across languages, etc.). This ensures accuracy and maintains the "single source of truth" principle for original language data.

Resolving Vector Dimension Mismatch (768 vs 1024) in Embeddings

The bible_db contains two sets of verse embeddings with different dimensions – one 768-dimensional (likely from an earlier model) and one 1024-dimensional (from a newer model) [25]. This mismatch must be addressed to avoid errors and ensure consistent semantic search. **Best practice is to standardize on a single embedding size** across the application, if possible. Mixing dimensions leads to complexity (a query vector from a 768-dim model can't be directly compared to 1024-dim vectors, and vice versa). The ideal solution is to **choose one embedding model** (e.g. if the newer 1024-dimension model is preferred for accuracy, re-embed the older data with that model; or if sticking with 768 for performance, down-sample the 1024 vectors). Standardizing ensures all vectors live in the same vector space.

If re-embedding all verses with one model is infeasible (due to time or cost), there are a couple of other strategies: one could *maintain separate indexes* for 768 and 1024 vectors and query them separately, but this complicates the search workflow and might confuse users. Alternatively, one might attempt to mathematically **project or pad** vectors from one dimension to the other – e.g. pad 768-dim vectors with 256 zeros to make them 1024-length. Zero-padding is simple and preserves the original vector components, but it changes the magnitude of vectors and could skew distance calculations (and it's not grounded in a known transformation, it's more of a hack). Conversely, truncating a 1024 vector to 768 (by dropping 256 dimensions) would lose information and isn't advisable unless one knows those extra dimensions are negligible. **Interpolating** or resampling vector dimensions (in the sense of some learned linear mapping) is an advanced approach – one could train a linear projection from 1024→768 or vice versa using a sample of data. This can align the spaces to some extent, but it introduces an extra layer of computation and potential error.

Given the system's emphasis on correctness and simplicity, the safest approach is to **standardize**: pick the target dimension and regenerate or adjust the data to match. In practice, if the 1024-dim embeddings are from a superior model (like a larger transformer) and the system can move forward with those, you might update the `bible.verses` table to have 1024-dim vectors (running a one-time migration to recompute them) and deprecate the old 768-dim field. This would eliminate the mismatch entirely. If both need to be kept (for comparison or versioning reasons), then clearly separate their use in code: e.g. use the 1024-dim for new features, and maybe remove the 768 from active use. The **project documentation** already flags this mismatch [26], so resolving it should be part of the design. In summary, **do not attempt on-the-fly dimension interpolation in production** (that could introduce subtle bugs); instead, unify the embedding dimensions either by re-embedding or by explicitly configuring separate search flows. The simplest best practice: use one embedding size everywhere and update the DB or code accordingly, ensuring vector math compatibility and easier maintenance.

Cross-Language AI-Noun Discovery and LLM Usage

Adapting the AI-driven noun discovery to work across languages requires careful design to maintain **correctness**. The current AI-noun discovery likely uses a language model (possibly fine-tuned on theology/biblical text) to identify important nouns or concepts in a passage. When extending this to be cross-language (i.e. to find related nouns between Hebrew and Greek texts), we should ensure the LLM is *only used in a controlled, "metadata-only" fashion*. By **"LLM = metadata only,"** we mean the LLM should not introduce new factual content or unverified links; it should only operate on data that we already have from the DB or serve in an assisting role (like classifying or formatting) [27]. This principle is explicitly stated as a core rule: all biblical content comes from the database, and LMs never invent or contribute new facts outside of that [27].

For cross-language noun discovery, one robust strategy is to ground the process in the **existing lexicon and embeddings**. For example, if you find a significant noun in a Hebrew verse (using the Hebrew text and lexicon data), to find a Greek equivalent you might use the English gloss or Strong's number to search the Greek corpus for matches. This can be done via the database (e.g. find Greek entries with the same Strong's root if applicable, or search by similar gloss), or via vector similarity on definitions. The LLM's role here could be to *explain* the connection or to rank the relevance of candidates, but it should not hallucinate connections. For instance, the system could retrieve a list of candidate Greek words related to a Hebrew concept (via a cross-reference table or vector semantic similarity), and then have the LLM choose which one is theologically most significant *based on definitions provided*. In doing so, the LLM would only use the **metadata given to it (glosses, definitions, verse contexts)** and not any outside knowledge. This aligns with the design where **LM explanations are DB-grounded and formatting-only** [28].

Another approach is to utilize **theology-specific LMs** only as a final step to articulate the findings. For example, after the module identifies that Hebrew *chesed* (חסד, loving-kindness) corresponds to Greek *eleos* (ἔλεος, mercy) via lexicon links, an LLM could be prompted to provide a short explanation or insight – but even then, the prompt should supply the relevant DB-derived info (e.g. "Hebrew חסד is translated as ἔλεος in Greek; explain this connection") so that the model isn't freewheeling. Essentially, the LLM can enrich the output *presentation* but the **discovery logic must remain deterministic and data-driven**. Techniques like verifying any LLM-suggested noun against the actual scripture text or lexicon ensure correctness. For example, if the LLM (guided by a theology model) suggests a concept link, the system should cross-verify that suggestion by checking if that noun actually appears in the relevant corpus or if the Strong's numbers align, etc., before accepting it.

In summary, adapt the noun discovery by anchoring it in bilingual lexicon data and possibly cross-language embeddings (which are under our control), and use the LLM only to assist with ranking or explaining those results – never to fabricate links. Maintain a clear rule: **the LLM does not introduce new nouns or translations that the database hasn't confirmed**. This will keep the cross-language AI noun discovery solid and trustworthy. It follows the project's doctrinal guideline that AI is used for "formatting only, never inventing content" [27], which is crucial in a theological context where accuracy is paramount.

Ensuring Gematria Module Determinism and Decoupling

The gematria module should be designed to be **deterministic, minimal, and independent** of any runtime orchestration or external state. *Deterministic* means that given the same input (biblical text or word), it

always produces the same numeric result with no randomness or AI variability. This is straightforward to achieve since gematria is essentially a pure function (sum of letter values); just avoid any nondeterministic processes (no LLM calls, no external API calls) inside the gematria calculations. The implementation can be as simple as a lookup table of letter→value and a sum routine – this guarantees repeatability and makes it easy to test. In fact, the module should have comprehensive unit tests verifying that known words produce known values (e.g. check that "913 = "בראשית, etc.), reinforcing determinism.

Keeping it **minimal** implies the gematria logic should have as few dependencies as possible – ideally just the mapping data and maybe the Unicode normalization utils. It should not depend on heavy parts of the system like database connections, the orchestration framework, or large libraries. This aligns with the plan's boundary that the Gematria module be *pure Bible logic with minimal dependencies, not tied to Gemantria.v2 infrastructure* [29] . By isolating it, you make it easier to maintain and possibly reuse in other contexts (and you reduce the risk of it breaking when other parts of the system change).

To keep it **decoupled from the runtime orchestrator**, the gematria module should be a self-contained library that the orchestrator can call, rather than the module reaching out. For example, the orchestrator (or a flow in the BibleScholar app) can pass a verse or word into a gematria function and get back results, but the gematria code itself shouldn't, say, log into the control-plane or make decisions about how to route flows. This separation of concerns follows the control-plane vs. data-plane design: gematria is data-plane (pure computation), while the orchestrator is control-plane (deciding when to call gematria, logging, etc.). In practice, this means **no global state or side effects** in the gematria functions. They should not, for instance, write to a database or file; they shouldn't even need to know *which* verse they are processing beyond the letters themselves. They certainly should not incorporate any LM calls or prompt logic. The module can be initialized with static data (like the mapping tables for Hebrew and Greek, perhaps loaded from a JSON or defined in code), and after that everything is just calculation.

By adhering to these principles, the gematria module remains **predictable and robust**. It can be treated as a black box by the rest of the system: given text → returns numbers/patterns. This also aids in testing (you can test it with no DB or network needed) and in performance (the calculations are trivial CPU-wise). The project documentation underscores that gematria logic should be extracted into a standalone module with minimal coupling [29] . Implementing it as a pure, deterministic library ensures that the overall architecture stays clean – the control-plane orchestrator can orchestrate, and the gematria module will do its job without needing to be entangled with orchestration logic. This fulfills the goal of a **deterministic, minimal Gematria module** that can evolve or be reused independently, all while integrating seamlessly into the larger multi-language analysis system via defined adapter interfaces.

Sources:

- Greek isopsephy letter values and obsolete letters [2] [3] ; standard mapping verification [5] .
- Modern Greek numeral notation (stigma/στ) [4] .
- Hebrew gematria methods (Mispar Hechrachi, Gadol, Katan, Shemi) [11] [13] [15] [16] ; final-letter values usage [14] .
- Unicode normalization for Greek and case-folding (remove accents, normalize sigma) [7] [8] ; Hebrew normalization pipeline rule [6] .
- Ketiv/Qere handling strategy in design (Ketiv primary, variants recorded) [17] .
- Lexicon adapter and morphology code explanation tables [21] [20] .
- Cross-language flow use of lexicon lemma field [22] .

- Embeddings dimension mismatch noted in DB (768 vs 1024) [25].
- Policy on LM usage (DB-sourced answers only, LMs for formatting) [27] [28].
- Module decoupling and purity guidelines [29].

---

[1] [6] [17] [29] AGENTPM_GEMATRIA_MODULE_PLAN.md
file://file-SJLapmCS2NUDtQSaVjgiyM

[2] [3] [4] [5] Isopsephy - Wikipedia
https://en.wikipedia.org/wiki/Isopsephy

[7] [8] regex - Javascript - normalize accented greek characters - Stack Overflow
https://stackoverflow.com/questions/23346506/javascript-normalize-accented-greek-characters

[9] [10] J. K. Tauber
https://jktauber.com/articles/python-unicode-ancient-greek/

[11] [13] [15] [16] TorahCalc | Explanations of Gematria Methods with Charts
https://www.torahcalc.com/info/gematria

[12] [14] Four Methods for Doing Gematria - GalEinai
https://inner.org/gematria/fourways.php

[18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28] BIBLESCHOLAR_MIGRATION_PLAN.md
file://file-FAB7BXYNZ1R1Mw5DmZmUkR