# Visual Bypass

## Reducing Token-Tax through Interface-Level Design

ID3000: Graduation project

Prabhanshu Sharma

TU Delft

# Visual Bypass

## Reducing Token-Tax through Interface-Level Design

by

Prabhanshu Sharma

Prabhanshu
Sharma

| | |
|---|---|
| Responsible Supervisor: | E. Niforatos |
| Daily Supervisor: | S. Degachi |
| Project Duration: | February, 2026 - August, 2026 |
| Faculty: | Faculty of Industrial Design Engineering, Delft |

Cover: Query kiviat visuals from the testing phase of design.

TU Delft

# Preface

*This thesis began as a question about tokens and ended up, somewhere along the way, as a question about fairness. About who gets to ask for help easily, and who has to pay more, wait longer, or say less, simply because of the language they think in. I did not expect a project about tokenizers to become a project I cared about this personally, but it did, and I suspect that is true of most work worth finishing.*

*My first thanks go to my supervisors, Evangelos Niforatos and Shatha Degachi, for their support, their feedback, and above all their patience. With a project that changed shape more than once, and with a researcher who needed the space to let it. Every sharper argument in this report exists because one of you asked a harder question first. To my friends: thank you for the constant support, the encouragement, and the belief that this would come together even in the stretches when I did not believe it myself. That kind of steadiness is easy to underestimate and impossible to do without.*

*Some of the thinking in this report did not happen at a desk. It happened outdoors, in the kind of quiet that nature is uniquely good at providing, where a system like an ecosystem, a coastline, a forest, turns out to be a better teacher of interdependence than any diagram I could draw. Some of it happened at hours I am not proud of, in the particular clarity that only shows up at 3 a.m. and is gone by breakfast. Both were, in their own way, necessary.*

*Two things I love outside of this work ended up inside it anyway, whether I planned that or not: a long-held love of geometry and visual form, which is most of the reason this thesis ends in a radar chart instead of a form full of checkboxes; and a belief, held for longer still, that equity is not a feature to add later but a constraint to design with from the start. If this report has a single throughline, it is an attempt to hold both of those at once.*

*I am finishing this at a moment when AI is becoming genuine infrastructure for how people around the world find help, ask questions, and make sense of what is happening to them. I would like to believe, and this thesis is, in its way, a small bet on the idea that the world can still become a more coherent, more equitable global community than the one we have today, and that the interfaces we choose to build are one of the places that outcome actually gets decided.*

*Prabhanshu Sharma*
*Rijswijk, August 2026*

# Summary

*Every AI query passes through a system: a training corpus shapes a tokenizer's vocabulary, that vocabulary is fixed inside a model, an API meters and prices access to it, a business layer wraps it in its own instructions, and only then does an interface let a user form their question. This thesis takes Donella Meadows's framing of* leverage points*, which suggests places in a complex system where a small, well-chosen intervention produces disproportionate effects, as its lens, applied to one dysfunction in that stack: a tokenizer's vocabulary, calibrated to whichever language dominates its training data, fragments non-English queries into more tokens than English ones for identical meaning. This* token tax *means two people asking the same health question pay a different price purely as a function of language. Every mitigation proposed so far, including expanding a tokenizer's vocabulary, compressing an already-written prompt, caching a repeated prefix, intervenes at the tokenizer or business layer; none touches the layer where a query is first formed. This thesis treats that interface layer as an unexamined leverage point and tests it directly, in the domain where the tax bites hardest: personal, multilingual consumer health questions.*

*The work proceeds in two movements. The first characterises the problem at its source: benchmarking real consumer health questions across English, Hindi, and Arabic, across three commercial models, confirms the tax is real, structural, and consistent in direction regardless of tokenizer, therefore a stable property of the model layer. The second tests whether the interface layer can counterweight it: MedQuery, a visual, non-verbal query interface built on a taxonomy of intents, open keywords, and ordinal parameters expressed through direct manipulation of a Kiviat diagram, replaces natural-language formulation with a compact, largely language-independent encoding, evaluated against free-text prompting across the same models and languages.*

*What this second study found is the clear demonstration of what a systems lens is for. The interface reduced token cost for some models, and reversed into a cost* increase *for another therefore, the largest effect observed anywhere in this analysis. Tracing that reversal upward through the stack shows it did not originate at the interface layer: it originated one layer above, in how differently each provider's caching mechanism absorbed the structured system prompt. The interface behaved identically in every condition; what changed was a business-layer decision the designer cannot see or control without deliberately measuring for it. Response similarity between conditions stayed stable and moderate throughout, consistent with the interface reshaping emphasis rather than correctness; token savings, where they occurred, did not imply faster responses, suggesting that cost and speed are governed by separate mechanisms within the same stack.*

*The conclusion is therefore not simply that interface design is, or is not, an effective lever against the token tax. It is that leverage exercised at one layer of a multi-layer system cannot be evaluated, or trusted, from that layer alone; it must be verified against the layers it depends on, empirically and per provider, as standing practice rather than a one-time decision. That is both this thesis's central finding and its methodological argument: the interface layer is real leverage, and knowing that required looking past the interface entirely.*

# Nomenclature

*This section lists the abbreviations, statistical symbols, and technical terms used throughout this report, for reference before the main text.*

## Abbreviations

| Abbreviation | Definition |
|---|---|
| ABSA | International Standard Atmosphere |
| ANCOVA | Analysis of Covariance |
| ANOVA | Analysis of Variance |
| API | Application Programming Interface |
| BPE | Byte-Pair Encoding |
| CHQ | Consumer Health Question |
| CV | Coefficient of Variation |
| GH | Games–Howell (post-hoc test) |
| HREC | Human Research Ethics Committee |
| HSD | Honestly Significant Difference (Tukey's post-hoc test) |
| IDE | (Faculty of) Industrial Design Engineering, TU Delft |
| JASP | Jeffreys's Amazing Statistics Program (the statistical software used for analysis) |
| KV cache | Key-Value cache (attention-state cache used during LLM inference) |
| LaBSE | Language-agnostic BERT Sentence Embedding |
| LLM | Large Language Model |
| MeQsum | Medical Question Summarization (the consumer health question dataset used throughout this thesis) |
| NLU | Natural Language Understanding |
| RIAS | Roter Interaction Analysis System |
| RQ | Research Question |
| UI/UX | User Interface / User Experience |

## Symbols

| Symbol | Definition | Unit |
|---|---|---|
| $df$ | Degrees of freedom. | – |
| $F$ | The F-statistic, the ratio of explained to unexplained variance tested in an ANOVA. | – |
| $M$ | Mean. | – |
| $MD$ | Mean difference, as reported in post-hoc pairwise comparisons. | – |
| MS | Mean Square (SS divided by df). | – |
| $N$ | Sample size (number of observations in a given cell or analysis). | Count |
| $p$ | The p-value; the probability of observing a result at least this extreme if the null hypothesis were true. Reported throughout at the conventional $\alpha = .05$ threshold. | Probability |

| Symbol | Definition | Unit |
|---|---|---|
| SD | Standard deviation. | – |
| SE | Standard error. | – |
| SS | Sum of Squares. | – |
| $t$ | The t-statistic, reported for individual pairwise post-hoc comparisons. | – |
| $\eta_p^2$ | Partial eta-squared; the proportion of variance in the dependent variable attributable to a given factor, holding other factors constant. Interpreted throughout using conventional small ($\approx 0.01$), medium ($\approx 0.06$), and large ($\approx 0.14$) benchmarks. | Proportion |
| $\omega^2$ | Omega-squared; an effect-size estimate similar to $\eta_p^2$ but with a smaller upward bias in unbalanced designs, reported alongside it throughout. | Proportion |

# Glossary of terms

**Token** The sub-word unit an LLM's tokenizer decomposes text into before processing; the unit compute, cost, and context-window budget are measured in (Section 2.2).

**Tokenizer** The component that converts raw text input into the sequence of tokens a model processes, with a vocabulary fixed at training time (Section 2.2).

**Byte-Pair Encoding (BPE)** The sub-word tokenization scheme underlying most tokenizers used in this thesis, which builds its vocabulary by merging frequently co-occurring character sequences from the training corpus (Section 2.2.1).

**Token fertility** The average number of tokens a tokenizer produces per word or character for a given language; the metric used to quantify the token tax (Section 2.2.1).

**Token tax** The disproportionate computational and financial cost incurred, purely as a function of language, for semantically equivalent input (Section 2.2.1).

**Input tokens** The raw token count of a submitted query, as returned directly by a model provider's API; the primary dependent variable for RQ1 (Section 3.1.1).

**Processed tokens** Input tokens minus cached tokens; the fair cost-comparison metric used for RQ2, since it accounts for tokens a provider does not charge or compute in full once cached (Section 3.2.1).

**Prompt / context caching** A provider-side mechanism that reuses previously computed attention states for a repeated prompt prefix rather than recomputing them, reducing compute cost and, potentially, latency for the cached portion of a call (Section 2.3).

**System prompt** The fixed instructional text prepended to every API call, invisible to the end user, that shapes how a model interprets and responds to the query that follows (Section 2.2).

**Condition A** In the RQ2 testing methodology, the MedQuery encoded-string interface condition (Section 3.2.5).

**Condition B** In the RQ2 testing methodology, the raw free-text prompting baseline condition (Section 3.2.5).

**Response similarity** The cosine similarity between the LaBSE sentence embeddings of a query's Condition A and Condition B responses; a measure of cross-condition agreement, not of correctness (Section 3.2.5).

**Intent** In the MedQuery taxonomy, the label describing what a user is fundamentally trying to accomplish with a query (e.g. Information Seeking, Symptom Checking), distinct from the query's content parameters (Section 3.2.3).

**FELT axes** Ordinal (0–5) parameters in the MedQuery taxonomy capturing self-reported experiential intensity: Symptom Severity, Emotional Distress, Duration (Section 3.2.3).

**FOCUS axes** Ordinal (0–3) parameters in the MedQuery taxonomy capturing topical emphasis independent of felt intensity, e.g. Pharmacology, Prognosis, Cost (Section 3.2.3).

**Keywords** The open, untagged, free-text field in the MedQuery taxonomy capturing nominal entities such as diagnoses or medications (Section 3.2.3).

**Coverage** In the inductive coding procedure, the proportion of a query's semantic clauses successfully mapped to a taxonomy category; queries at or above 80% coverage were considered adequately represented by the taxonomy (Section 3.2.3).

**Kiviat diagram** Also called a radar chart; a multi-axis visualisation in which each axis radiates from a shared centre, used in this thesis as a direct-manipulation input control rather than an output display (Section 3.2.4).

**Nominal scale** A measurement scale whose categories differ in kind but carry no inherent order (Stevens, 1946); the basis for treating Keywords as untagged free text (Section 2.5).

**Ordinal scale** A measurement scale whose categories can be meaningfully ranked; the basis for the FELT and FOCUS axes (Section 2.5).

**Stakeholder cascade** The five-layer chain: model developers, API/business layer, application/UI design, end users, downstream effects, and through which an AI query passes (Section 2.1).

**Leverage point** Following Meadows (2008), a place within a complex system where a small, well-targeted intervention can produce disproportionately large system-wide effects; the framing under which the interface layer is treated as a candidate intervention point throughout this thesis.

# Contents

# List of Figures

# List of Tables

# 1

# Introduction

## 1.1. A socio-technical system

When someone types a question into an AI assistant, the exchange looks simple: a question in, an answer out. Underneath, that single exchange passes through a chain of actors and infrastructure. The interface the person types into, the business logic and system prompts that shape the request, the model that processes it, the training data and tokenizer that determine how the model reads language in the first place (Ahia et al., 2023). Each layer was designed by a different party, for a different purpose, and each one shapes what the user ultimately experiences (Pool et al., 2024). This thesis treats AI-mediated health information-seeking as exactly this kind of socio-technical system, and adopts a systems-thinking lens throughout. Problems that appear at the interface are often symptoms of decisions made several layers upstream, and interventions at one layer can have effects, whether intended or not, at every other layer (Meadows, 2008; Sambasivan et al., 2021). This framing is what allows this thesis to ask an interface-design question, addressed later in this thesis, to speak to a much larger infrastructural inequity (Bieniek et al., 2024; Solatorio et al., 2024).

## 1.2. Global adoption of AI

AI assistants are used worldwide, across hundreds of languages, but the infrastructure they run on was not built evenly across those languages (Misra et al., 2025). Large language models(LLMs) are trained on corpora that consist of a mix of languages. Recent research shows that most of it is in English and a handful of other high-resource languages. Moreover, the tools built on top of them - tokenizers, vocabularies, evaluation benchmarks - are directly influenced by the training. (Ahia et al., 2023; Goyal et al., 2021; Lundin et al., 2026).Ahia et al. (2023) shows that the number of tokens processed for a query depends on the input language. The result is not merely a difference in fluency or output quality; it extends to the basic mechanics of how a model processes a user's input, as the next section explains (Petrov et al., 2023; Solatorio et al., 2024).

## 1.3. Tokenization as a cost mechanism

Before a language model can process a query, it breaks the text into tokens that are the sub-word units the model actually process, reads, and reasons over. This process is done by a tokenizer. Tokens are also the unit providers use to meter and bill a query, and the unit that determines how much of a model's limited context window a query consumes (Ahia et al., 2023). A tokenizer is essentially a lookup table: it was built by observing patterns in a training corpus, and it compresses frequently seen patterns into fewer tokens. Since that training corpus over-represents English, English text is tokenized efficiently, while the same sentence in a lower-resource language, with different scripts, morphology, and word-formation patterns, often needs substantially more tokens to represent the identical meaning (Ahia et al., 2023; Toraman et al., 2023). Recent work has quantified this gap directly: depending on the language pair, the same content can require up to 15 times as many tokens to encode (Petrov et al., 2023). Because cost, latency, and available context all scale with token count, this is not a cosmetic

difference (Lundin et al., 2026). It behaves like a penalty: two users asking the same question, in the same intent, pay a different price and get a different effective budget, purely as a function of which language they typed in (Petrov et al., 2023; Solatorio et al., 2024).

## 1.4. The gap

Existing efforts to soften this penalty consist of context caching, prompt compression, expanding tokenizer vocabularies to cover more languages. These practices operate almost entirely at the model or infrastructure layer (detailed in Chapter 2) (Ahia et al., 2023; Petrov et al., 2023). What has been left comparatively unexamined is the layer closest to the user: the interface itself, where a query is first formed (Bieniek et al., 2024). If tokenization cost is partly a function of *how* a query is expressed, not just *what language* it is expressed in (Ben Abacha & Demner-Fushman, 2019), then interface design becomes a potential lever for mitigating the tokenization penalty. In the LLM-focused literature, this has received little attention because the problem has largely been framed as an NLP or infrastructure issue rather than an HCI one (Kim et al., 2024). This is the white space this thesis addresses. This thesis refers to that disproportionate cost as the *token tax*, the term used throughout the remainder of this report.

## 1.5. Research questions

To test the hypothesis of leveraging an interface layer to reduce the token tax, this thesis proceeds in two stages: first diagnosing the scale of the problem, then testing an interface-layer intervention against it.

- **RQ1.** Does input token count differ significantly across languages (English, Hindi, Arabic) and different models, for equivalent consumer health queries?
- **RQ2.** Can an interface intervention reduce input token cost relative to raw-text prompting, across models and languages, and at what cost to response similarity and latency?

## 1.6. Structure of the report

This report tells the story in four movements. It starts by widening the systemic thinking lens: Chapter 2 traces the token tax from where it originates. A training corpus, a tokenizer's vocabulary, through every layer of the AI service stack it passes on the way to a user, and closes on the specific gap this thesis is trying to address: an interface layer that every prior mitigation walked past. Chapter 3 then does two things in sequence, matching the two research questions above: first measuring the size of the problem directly, benchmarking real health questions across languages and models, and then designing and testing an answer to it. Chapter 4 reports what happened when that design met real models. Chapter 5 is where those results are read back through the systems lens this introduction opened with. Chapter 6 and 7 close the report with its limitations and conclusions.

# 2

# Literature Review

This review is organised through the systems-thinking lens introduced in Section 1.1 (Meadows, 2008): it first defines the system this thesis intervenes in and locates where cost accrues within it (2.1), then works through the mechanics and magnitude of the token tax (2.2–2.3), before turning to the domain and interaction-design literature that motivates the specific intervention tested in Chapter 3 (2.4–2.6), and closing on the knowledge gap this thesis fills (2.7).

## 2.1. System boundary and the stakeholder cascade

In the context of an AI health query, it also does not travel through a single system; it travels through a chain of five actors, each with a different mandate, and each shaping the token tax in a different way (Figure 2.1) (Pool et al., 2024). **Model developers** train the base model and, in doing so, fix the tokenizer's vocabulary. This decision is further examined in Section 2.2, determines how efficiently any given language can later be expressed (Ahia et al., 2023; Petrov et al., 2023). The **API/business layer** sits above this: it sets per-token pricing, decides whether and how context caching is offered, and defines the system prompts that wrap every user query before it reaches the model (Ahia et al., 2023; Lundin et al., 2026). The **application/UI layer** is where an end user's intent is first translated into a submittable query, which could be a text box, a form, or a set of buttons, and it is the layer this thesis intervenes at (Bieniek et al., 2024). **End users** sit downstream of all three: whatever cost, latency, or accuracy consequences were set upstream are the ones they experience, without necessarily seeing why (Lundin et al., 2026; Solatorio et al., 2024). Finally, those individual experiences aggregate into **downstream effects** on healthcare access and equity, that is, who can afford to ask a detailed question, in which language, and how quickly they get an answer (Solatorio et al., 2024). A feedback loop closes the cascade: aggregated usage and interaction data eventually feed back into the training corpora of future models, meaning today's cascade partly determines tomorrow's tokenizer(Solatorio et al., 2024).

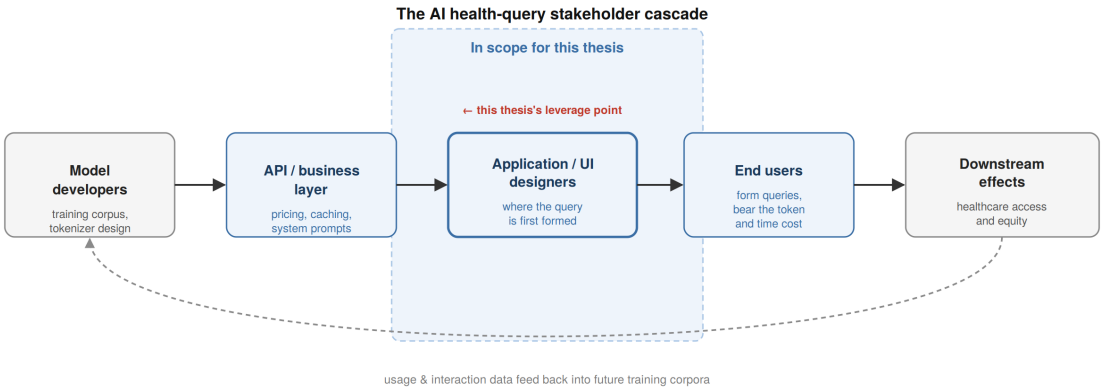



**Figure 2.1:** The stakeholder cascade through which an AI health query passes, from model training to downstream equity effects. The application/UI layer (highlighted) is this thesis's point of intervention.

**Scope.** This thesis is bounded to the application/UI layer of this cascade. It does not attempt to retrain a model, redesign a tokenizer, or change API pricing policy, all upstream, infrastructure-level interventions that lie outside an individual interface designer's control and outside this project's resources. What it does ask is whether the layer closest to the user, the one layer in this cascade that an interface designer can act on directly, can absorb some of the cost that upstream layers impose, before it reaches the user. The remainder of this chapter builds the case for why that layer, specifically, has been left comparatively unexamined.

## 2.2. AI service architecture and the token as the metric unit

Section 2.2 introduced the token, at a mechanical level, as the sub-word unit a tokenizer breaks text into. What matters for this thesis is what the token *does* once it exists: it is the single unit that every layer of the cascade in Figure 2.1 measures, prices, or budgets by (Ahia et al., 2023). Compute and energy consumption during inference scale directly with the number of tokens a model processes (Lundin et al., 2026; Solatorio et al., 2024; Strubell et al., 2019), so the token is not only a billing unit but an environmental one. Tracing how a token accrues cost requires walking the stack it passes through (Figure F.1).

At the base of the stack, the **training corpus** determines the tokenizer's vocabulary: the sub-word units it later merges into single tokens are the ones that were frequent in that corpus (Ahia et al., 2023). The **model and tokenizer** inherit this vocabulary permanently, as it is fixed at training time, not adjusted per query, and this is where the first cost accrues, since every token processed at inference draws compute and energy regardless of which language produced it (Lundin et al., 2026; Solatorio et al., 2024). Above the model, the **API layer** converts token count directly into price, and additionally fixes the size of the context window each query must fit inside (Ahia et al., 2023; Petrov et al., 2023). The **business/application layer** adds its own token overhead on top of the user's query; for example, system prompts, formatting instructions, and few-shot examples are all invisible tokens a business bears on every call before the user's own input is even read (Ahia et al., 2023). Finally, the **UI/input surface** is where the literal content and length of the query is authored.It is the only layer in this stack where the number of tokens submitted is a direct function of interface design rather than infrastructure (Bieniek et al., 2024). A feedback loop closes the stack: aggregated usage data eventually informs the corpora future models are trained on, so today's UI-layer decisions can, over a long enough horizon, shape tomorrow's tokenizer vocabulary too (Solatorio et al., 2024).

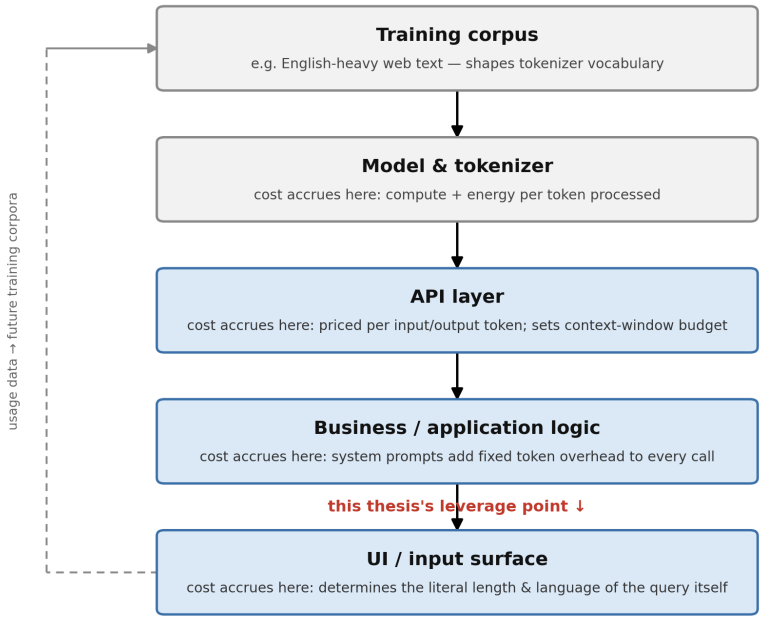



**Figure 2.2:** The layered AI service architecture. The token is the unit that links every layer, from the training corpus that calibrates the tokenizer to the UI surface that ultimately determines query length.

The mechanical link that carries into the next section is this: because the tokenizer's vocabulary (bottom of the stack) is calibrated to whichever language dominates the training corpus, and because every layer above it prices, budgets, or bills in that same tokenizer's units, any imbalance built into the vocabulary propagates upward through the entire stack, reaching the user as cost, latency, and reduced effective context, not as an abstract inefficiency, but as the concrete, language-dependent penalty examined next.

### 2.2.1. The token tax

The mechanism behind this penalty is straightforward once the tokenizer is understood as a lookup table rather than a linguistic rulebook. Most current tokenizers use variants of byte-pair encoding, which builds its vocabulary by repeatedly merging whichever character sequences occur most frequently in the training corpus into single units (Sennrich et al., 2016). A word that appeared constantly in that corpus which is generally overwhelmingly English, per Section 2.2, earns a compact, single-token representation (Ahia et al., 2023). A word in a script or morphology the corpus rarely contained is not "translated" into more tokens; it simply never earned a merged representation, so the tokenizer falls back to fragmenting it into smaller pieces (Figure 2.3)(Petrov et al., 2023; Toraman et al., 2023). The same concept, expressed with equal precision in two languages, can therefore cost a different number of tokens for a reason that has nothing to do with the concept's complexity and everything to do with which language happened to dominate the training data (Ahia et al., 2023; Petrov et al., 2023).

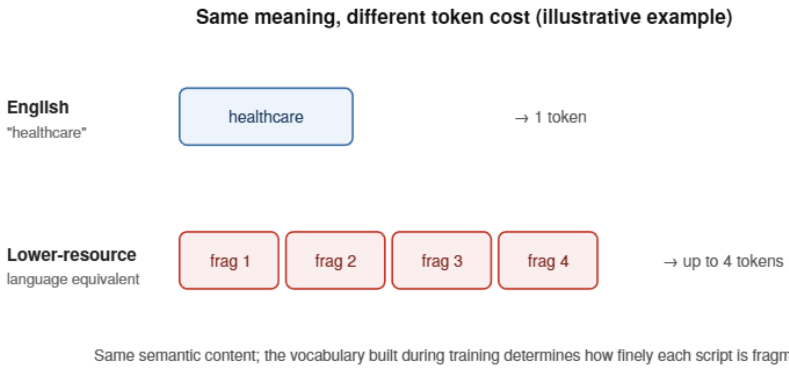



**Figure 2.3:** Illustrative example of the fragmentation mechanism: a word with a compact, single-token representation in a high-resource language may fragment into several tokens in a lower-resource one, for the same meaning.

The magnitude of this gap has been measured directly.Petrov et al. (2023) benchmarked tokenizer behaviour across a broad set of languages and tokenizer architectures that includes ones explicitly built for multilingual support, and found that the same content can require up to 15 times as many tokens in one language as another, with commercial API access at least 2.5 times more expensive for some language communities than for English speakers on equivalent content. Ahia et al. (2023) reached a related conclusion from the business side: analysing commercial API pricing and output quality across 22 typologically diverse languages, they found that speakers of a large share of supported languages are simultaneously overcharged for semantically equivalent input and receive lower-quality output. This double burden disproportionately falls on regions where the same API was already comparatively less affordable to begin with, reflecting systemic socio-economic disparities. The effect is not limited to a handful of edge-case languages: Toraman et al. (2023) documented a comparable fertility penalty for Turkish, a morphologically rich but Latin-script language, suggesting the disparity tracks morphological and script distance from the training-dominant language more than script family alone.

This is the term this thesis refers to as the token tax (Section 2.2): a disproportionate computational and financial cost, incurred purely as a function of language, for semantically equivalent input (Lundin et al., 2026). Its consequences compound at every layer walked in Section 2.2. For users, more tokens mean higher cost and latency per query, and since every model imposes a fixed context-window limit measured in tokens, not words, it becomes a smaller effective budget for expressing the same information need (Ahia et al., 2023; Petrov et al., 2023). For businesses, API pricing denominated in tokens means that a product serving non-English markets pays structurally more to deliver the same service, a cost that compounds across every query at scale and can widen, rather than narrow, existing economic gaps between high and low-resource language markets (Ahia et al., 2023; Korinek & Stiglitz, 2026; Solatorio et al., 2024). None of this requires malicious intent anywhere in the pipeline: the token tax is an emergent property of how tokenizers are built, not a deliberate design choice at any single layer. This is precisely what makes the token tax structural, and why the mitigation strategies reviewed next have concentrated on redesigning that pipeline rather than questioning whether the pipeline is the only place a fix can happen (Meadows, 2008; Petrov et al., 2023).

## 2.3. Current mitigation practices

If the token tax originates in the tokenizer's vocabulary (Section 2.2.1), the most direct response is to fix the vocabulary itself (Ahia et al., 2023; Petrov et al., 2023). Proposed and deployed mitigations cluster into three approaches, all operating at or below the model layer of Figure F.1. **Vocabulary expansion** adds more non-English sub-word units to the tokenizer so that scripts like Devanagari or Arabic are represented as efficiently as English, which could be a direct fix, but one that requires retraining or, at minimum, re-fitting the tokenizer, and is only available to whoever controls the model (Ahia et al., 2023). **Language-adaptive tokenization** takes this further, allowing the merge rules themselves to vary by detected input language rather than sharing one global vocabulary across all languages. This makes it more targeted, but architecturally more complex, and again a model-provider-level change

(Petrov et al., 2023). **Prompt compression** operates one layer up: rather than changing the tokenizer, frameworks such as LLMLingua (Jiang et al., 2023) algorithmically shorten a prompt's text *after* a user has already written it, discarding low-information tokens while trying to preserve meaning.

What unites all three approaches is where they intervene: after the user's query already exists as natural language text, and before or during the moment the tokenizer processes it. Vocabulary expansion and adaptive tokenization act on the tokenizer (Ahia et al., 2023; Petrov et al., 2023); prompt compression acts on the text the tokenizer receives (Jiang et al., 2023).

A fourth lever operates one layer further up the stack, at the business/application layer rather than the model layer: **prompt and context caching**(Gim et al., 2024; Kwon et al., 2023). During inference, a model computes key-value (KV) attention states for every token it processes; when consecutive requests share an identical prefix, most commonly a fixed system prompt, re-sent on every call, a provider can cache and reuse those attention states rather than recomputing them, reducing both compute cost and latency for the shared portion of the prompt (Gim et al., 2024; Kwon et al., 2023). In principle, this means a fixed, unchanging system prompt (Section 2.2) need not cost the same on every call: once cached, only the novel, per-query portion of the input requires full computation (Gim et al., 2024). In practice, caching behaviour is provider- and architecture-specific rather than standardised; whether a given prefix is cached, for how long, and how much of its cost is actually discounted varies across API providers and is not something an application developer can fully observe or guarantee from outside (Ahia et al., 2023).

Neither type of intervention, tokenizer-level or caching-level, touches the step that precedes all of them: the moment a user first formulates their query (Bieniek et al., 2024). A prompt-compression algorithm can only shorten a sentence that has already been written in full (Jiang et al., 2023); a cache can only discount a prefix that recurs identically across calls (Gim et al., 2024; Kwon et al., 2023); neither reduces the extraneous formulation effort the user incurs in writing the query in the first place (Peng et al., 2024; Sweller, 1988). This is the specific gap this thesis positions itself in: not a fifth way to fix the tokenizer, compress existing text, or optimise caching, but an argument that the interface layer, which is upstream of all of them and has been left out of the mitigation conversation entirely (Kim et al., 2024).

## 2.4. AI in healthcare and consumer health information seeking

Health information-seeking has moved substantially onto conversational AI. Where this behaviour once meant searching a static web page, users increasingly pose their questions directly to an LLM-based chatbot and receive a personalised, dialogic response (Yun, Bickmore, et al., 2025). This shift matters for the token tax specifically: a search engine query is typically a handful of keywords, but a conversational health query is a full natural-language description of a personal, often emotionally weighted, experience which makes it exactly the query type most exposed to the fragmentation mechanism described in Section 2.2.1, since it is long, descriptive, and cannot easily be reduced to a short keyword string without losing the information a model needs to respond appropriately (Ben Abacha & Demner-Fushman, 2019; Yang et al., 2024).

This thesis draws its empirical substrate from consumer health questions (CHQs): naturally occurring questions submitted by members of the public to a health information service, as distinct from clinician-authored or exam-style medical questions.Ben Abacha and Demner-Fushman (2019) compiled and released MeQSum, a benchmark corpus of CHQs paired with expert-written summaries, developed specifically because such questions are typically long, colloquially phrased, and layered with peripheral detail for example, context, history, and emotional framing which is around the core informational need. This structure is precisely what makes CHQs a demanding case for both natural-language input generation and, by extension, token cost: a user describing a symptom rarely states only the clinically relevant facts, but embeds them within a fuller, more narrative account of their situation (Ben Abacha & Demner-Fushman, 2019). Kilicoglu et al. (2018) take this a step further, showing that CHQs can be semantically decomposed into structured components like entities, intents, and frames, via annotation schemes originally developed for clinical and biomedical text. Their work establishes that a CHQ's informational content is not, in principle, inseparable from its natural-language form: the same question can be represented as a structured semantic object without loss of the information a downstream

system needs (Kilicoglu et al., 2018). This precedent is taken up directly in Section 2.5, which builds the specific decomposition this thesis' design intervention relies on.

Health is therefore a domain where three conditions this thesis needs converge: queries are personal and experiential rather than terse and keyword-like, making them disproportionately exposed to the token tax; a validated, ecologically representative dataset of such queries already exists (MeQSum); and prior work has already demonstrated that these queries can be meaningfully decomposed into structured components without discarding their communicative content. The next section builds on that last point directly.

## 2.5. Decomposing a query into structured parameters

Kilicoglu et al. (2018) establish that a CHQ *can* be decomposed into structured components without losing the information it carries. What that work does not settle is what kind of components those should be, or whether all of them should be treated the same way. This distinction matters directly for an interface that asks a user to specify their query through discrete, adjustable inputs rather than a sentence: some things a person needs to convey are categorical labels, and others are matters of degree, and conflating the two produces an interface that is either too rigid or too vague.

Measurement theory gives this distinction a formal basis. Stevens (1946) separates nominal scales, in which categories differ in kind but carry no inherent order; for example, a diagnosis is not ”more” or ”less” than another diagnosis, from ordinal scales, in which categories can be meaningfully ranked, such as a pain rating of 4 out of 5 exceeding a rating of 2 (Yang et al., 2024). A named clinical entity (a diagnosis, medication, or procedure mentioned in a query) is nominal: the set of possible values is open-ended, and no value is inherently greater than another (Kilicoglu et al., 2018). A symptom's severity, or how long it has persisted, is ordinal: bounded, rankable, and expressible as a position on a scale. Therefore, named entities can be treated as open, untagged keywords, while experiential qualities can be treated as scaled, adjustable axes; these are not two arbitrary input styles, but two different measurement types requiring two different controls (Brooke, 1996). this motivates the conceptualisation discussed in Chapter 3.

Within the ordinal category, a further distinction is needed between two kinds of ”more.” Some axes describe how a symptom *feels* to the person experiencing it, which is a self-reported intensity, such as pain severity or emotional distress (Paruchuri et al., 2025). Others describe how much of the query is *about* a given topic and not a felt state at all, but a question's topical emphasis, such as how much a query concerns cost, prognosis, or treatment options (Paruchuri et al., 2025). Medical communication research offers partial grounding for the first kind: the Roter Interaction Analysis System (RIAS), built on Bales (1950) earlier framework for coding group interaction, separates affective, socio-emotional communication from task-focused, informational communication in clinical dialogue. This precedent supports treating a felt state like emotional distress as its own category; it grounds symptom severity and duration less directly, though both remain consistent with the broader convention of self-reported intensity scales used throughout patient-reported outcome measurement. For the second kind which is topical emphasis rather than felt intensity, an aspect-based sentiment analysis offers a closer structural parallel: the SemEval ABSA task formalises text annotation as an aspect category (a topic) paired with a separately scored attribute (Pontiki et al., 2014), a structure that maps onto scoring how much a query concerns a given theme without implying anything is felt about it.

Before determining what interface modality might capture this structure, it is worth making the decomposition itself concrete. A typical consumer health query is not a monolithic block of text; it is a layered construct of nominal entities and ordinal experiential states, compressed into a single, unstructured paragraph by the demands of natural-language writing. Table 2.1 decomposes a single representative query from MeQsum dataset (Ben Abacha & Demner-Fushman, 2019) , *”I was diagnosed with ncs about 10 yrs ago but little was known about it at the time. They said I needed surgery but refused due to the fact I didn't have any money. It has been left untreated all this time and is feeling worse. What can I do to get help with this? I am pleading for help, thank you.”* , against exactly this structure.

**Table 2.1:** Anatomy of a single consumer health query, decomposed into intent, nominal entities, and ordinal experiential states

| Layer | Parameter | Value | Textual evidence |
|---|---|---|---|
| Intent | Treatment Seeking | , | *"What can I do to get help with this?"* |
| Nominal (Keyword) | Diagnosis | "NCS" | *"diagnosed with ncs"* |
| Nominal (Keyword) | Procedure | "surgery" | *"They said I needed surgery"* |
| Nominal (Keyword) | Treatment status | "untreated" | *"left untreated all this time"* |
| Ordinal, FELT (0–5) | Duration (DUR) | 5 | *"about 10 yrs ago ... left untreated all this time"* |
| Ordinal , FELT (0–5) | Symptom Severity (SYM) | 4 | *"is feeling worse"* |
| Ordinal , FELT (0–5) | Emotional Distress (DIS) | 4 | *"I am pleading for help"* |
| Ordinal, FOCUS (0–3) | Cost (CST) | 3 | *"refused due to the fact I didn't have any money"* |
| Ordinal, FOCUS (0–3) | Referral (REF) | 3 | *"What can I do to get help with this?"* |
| Ordinal , FOCUS (0–3) | Prognosis (PRG) | 2 | *"is feeling worse"* (implied concern about trajectory) |

Two things are worth noting about this decomposition. First, none of the nine parameter values shown were stated by the author in anything resembling structured form; every one is inferred from a sentence written for a human reader, not a machine, which is exactly the formulation overhead Section 2.3 argued no existing mitigation addresses. Second, the same clause can carry more than one kind of information at once: "is feeling worse" supports both a moderate Symptom Severity score and an implied Prognosis concern, illustrating why a single free-text field cannot be cleanly separated into independent parameters without an explicit taxonomy to decompose it against. This is the decomposition Section 3.2 operationalises at scale, first through the inductive coding procedure described in Section 3.2.3, and then through the interface built to let a user supply this structure directly rather than have it inferred after the fact.

These precedents ground the *categorical* distinctions the design relies on, nominal versus ordinal, felt versus topical. The particular set of axes developed in Chapter 3 is assembled for this design problem specifically, and is best read as a novel synthesis of these adjacent frameworks rather than an instantiation drawn wholesale from any single one of them.

Having established that a query can be decomposed, and into what kinds of components, the remaining question is what kind of interface has historically been used to capture structured, multi-dimensional input of this kind, the subject of the next section.

## 2.6. History of Interface modalities

Every shift in dominant interface modality has changed the trade-off between how much effort it takes a user to express their intent and how much of that intent the system can reliably capture (Butler et al., 2025). Command-line interfaces required exact, memorised syntax, which is a high barrier to entry, but a compact and unambiguous one once learned (Bieniek et al., 2024; Butler et al., 2025). The graphical user interface, and specifically Shneiderman (1983) principle of direct manipulation, replaced typed syntax with visible objects and continuous, reversible actions (pointing, dragging, clicking) that remove the syntax barrier at the cost of confining the user to whatever actions the interface's designer chose to expose(Jørgensen & Myers, 2008). Structured, form-based input sits at a different point on this trade-off again: it sacrifices some of the GUI's open-ended flexibility in exchange for output that is predictable and directly parseable by a downstream system, which is precisely why forms remain the default for data collection in clinical and administrative contexts (Yang et al., 2024). Voice interfaces

reintroduced natural language as the primary input, trading the typing effort of text entry for the ambiguity of speech recognition and the loss of a persistent visual reference to what was said (Bieniek et al., 2024; Jørgensen & Myers, 2008).

The current generative-AI interface landscape has, on the whole, not continued this trajectory on the input side (Bieniek et al., 2024).Kim et al. (2024) taxonomy of UI/UX approaches for generative AI systems shows that design attention has concentrated overwhelmingly on how generated output is presented, refined, and iterated upon, rather than on how user intent is captured before generation begins; free-text entry remains the near-universal default for the input step itself, even as output-side interaction has grown considerably more sophisticated. This asymmetry is notable given the throughline of this section: every earlier modality shift changed the input side's cost/accessibility trade-off deliberately, yet the current generation of AI interfaces has largely left that side unexamined, evaluated historically for usability and learnability rather than for the token cost this thesis is concerned with (Ahia et al., 2023; Kim et al., 2024). That combination, an input side left underdeveloped, and never evaluated through a token-cost lens at all, is where the final gap this thesis addresses sits, and is made explicit next (Bieniek et al., 2024; Petrov et al., 2023).

## 2.7. Knowledge gap

Four threads from this chapter converge on the same point. The token tax is real, measurable, and structural: it originates in how a tokenizer's vocabulary is calibrated to its training corpus, and propagates upward through every layer of the service architecture, from compute cost to API pricing to the effective context budget a user has available (Sections 2.2–2.2.1). Every mitigation currently pursued for this problem- vocabulary expansion, adaptive tokenization, prompt compression, prompt caching- intervenes at the tokenizer or at text that has already been written; none of them touches the step that produces that text in the first place, the user's own formulation of their query (Section 2.3). Health queries are a domain where this unaddressed step matters most: they are personal, experiential, and long by nature, making them disproportionately exposed to the tax, while existing work shows they can be decomposed into structured components, categorical entities, and scaled, rankable qualities, without losing what they communicate (Sections 2.4–2.5). And the interface layer, historically the place where every prior shift in input modality renegotiated the trade-off between what a user must express and how easily a system can capture it, has in the current generation of AI tools been left comparatively undeveloped on the input side, and has never been evaluated specifically for its effect on token cost (Section 2.6).

Put together, these threads describe a gap rather than a series of separate observations: an unaddressed leverage point sits at the one layer of the AI service architecture (Figure 2.1) that an interface designer can act on directly, for a query type that is both especially exposed to the tax and already known to be decomposable, using interaction principles, direct manipulation, structured input, that HCI has applied to other problems for decades but not yet to this one. This thesis addresses that gap directly. RQ1 (Section 1.5) establishes the scale of the token tax for the specific language triad and domain this thesis works with, providing the empirical baseline the intervention is measured against; RQ2 tests whether a visual-first, structured-query interface, built on the decomposition established in Section 2.5, can reduce that cost without discarding what a free-text query would otherwise communicate. The methodology for both studies is developed next.

Table 2.2 consolidates the sources this chapter has drawn on, organised in the order they were introduced, as a single point of reference for the argument this chapter has built.

**Table 2.2:** Summary of key literature reviewed in this chapter

| Source | Focus | Key contribution | Role in this thesis |
|---|---|---|---|
| Meadows (2008) | Systems thinking | Leverage points: small interventions at the right place in a system produce disproportionate effects | Organising lens for the whole thesis |
| Strubell et al. (2019) | Compute/energy | Inference-time compute and energy scale with token count | Establishes tokens as an environmental, not only financial, unit |
| Sennrich et al. (2016) | Tokenization | Byte-pair encoding merges frequent training-data sequences into single tokens | Mechanism behind unequal tokenization across languages |
| Petrov et al. (2023) | Tokenizer fairness | Token counts differ up to 15× across languages; costs up to 2.5× higher for some communities | Primary magnitude evidence for the token tax |
| Ahia et al. (2023) | API cost/quality | Non-English speakers are simultaneously overcharged and under-served in output quality | Business-side evidence of the token tax's double burden |
| Toraman et al. (2023) | Turkish tokenization | Fertility penalty replicated for a morphologically rich, Latin-script language | Shows the tax is not script-family-specific |
| Korinek and Stiglitz (2026) | AI and development economics | AI productivity gains risk concentrating in high-income, English-dominant economies | Frames the token tax as an economic equity issue |
| Jiang et al. (2023) | Prompt compression | LLMLingua shortens already-written prompts algorithmically | Model-layer mitigation; motivates the interface-layer gap |
| Gim et al. (2024) and Kwon et al. (2023) | Prompt/KV caching | Repeated prefixes can be cached and reused, cutting compute for that portion | Establishes the caching mechanism central to the RQ2 finding |
| Yun, Bickmore, et al. (2025) | Health information-seeking | Health queries are increasingly posed directly to conversational AI | Motivates health as the thesis's query domain |
| Ben Abacha and Demner-Fushman (2019) | Consumer health questions | MeQSum: a validated corpus of real, naturally occurring health questions | Empirical substrate for both RQ1 and RQ2 |
| Kilicoglu et al. (2018) | CHQ semantic annotation | Consumer health questions can be decomposed into entities, intents, and frames | Precedent for decomposing queries without losing content |
| Stevens (1946) | Measurement theory | Distinguishes nominal (unordered) from ordinal (ranked) scales | Basis for the Keywords/FELT/FOCUS taxonomy split |
| Bales (1950); RIAS | Communication coding | Separates affective/socio-emotional from task-focused communication | Partial grounding for the FELT axes |
| Pontiki et al. (2014) | Aspect-based sentiment analysis | Aspect category paired with a separately scored attribute | Structural precedent for the FOCUS axes |
| Shneiderman (1983) | Direct manipulation | Continuous, visible, reversible actions reduce interaction overhead | Grounds the Kiviat interface as an input control |
| Kim et al. (2024) | Generative AI UI/UX | Design attention concentrates on output presentation, not input capture | Confirms the interface-layer gap this thesis targets |

# 3

# Methodology

This thesis is organised around two sequential empirical phases, one per research question. Section 3.1 describes the benchmarking study that answers RQ1: does the token tax exist, at what magnitude, for the specific language triad and query domain this thesis works with. Section 3.2 describes the design and evaluation study that answers RQ2: does a visual-first, structured-query interface reduce that cost, and at what trade-off to response similarity and latency. The first phase establishes the baseline that the second phase is measured against.

## 3.1. Benchmarking design (RQ1)

### 3.1.1. Hypotheses

The benchmarking study tests the null hypothesis that input token count does not differ significantly across languages or models for equivalent consumer health queries (H0), against the alternative that it does (H1). Two independent variables are named explicitly: **Language** (three levels: English, Hindi, Arabic) and **Model** (three levels: `gemini_25_flash_lite`, `gemini_31_flash_lite`, `gpt54`). The dependent variable is `input_tokens`, the raw token count returned by each provider's API for a given query.

### 3.1.2. Language selection

English serves as the baseline, consistent with its dominance in training corpora established in Section 2.2. Hindi and Arabic were selected specifically for their structural distance from English: both are written in non-Latin scripts (Devanagari and a right-to-left Perso-Arabic script, respectively), and both diverge from English morphologically, making them a demanding test of tokenizer behaviour rather than a near-equivalent comparison 2.2.1. Selecting two typologically distinct non-English languages, rather than one, also allows Chapter 4 to test whether the token tax is uniform across non-English languages or itself varies by language, a question Section 2.2.1 left open.

### 3.1.3. Model selection

Three commercial model conditions were benchmarked: `gemini_25_flash_lite` (`gemini-2.5-flash-lite`), `gemini_31_flash_lite` (`gemini-3.1-flash-lite-preview`), and `gpt54` (`gpt-5.4`). These span two independently developed tokenizer architectures, Google's Gemini family and OpenAI's GPT family, allowing any observed language effect to be checked against more than one tokenizer implementation, consistent with the mitigation-practice review in Section 2.3. Where a model's thinking mode could not be fully disabled, its thinking budget was fixed at the minimum valid value rather than swept as an experimental condition, keeping thinking-token generation constant rather than treating it as a third independent variable.

### 3.1.4. Dataset selection
Both empirical phases of this thesis draw on MeQSum (Ben Abacha & Demner-Fushman, 2019), a corpus of 1,000 naturally occurring consumer health questions submitted to a public health information service (introduced in Section 2.4). For the benchmarking phase, the full 1,000 queries are used. This sample size was chosen to provide adequate statistical power for a two-way factorial design with nine language×model cells, while remaining feasible within the API cost and rate-limit constraints available to the researcher.

### 3.1.5. Dataset preparation
MeQSum's queries are English-language by origin. Hindi and Arabic versions were produced via the DeepL translation API (Appendix A), applied to all 1,000 queries. Translated text may not be perfectly length-equivalent to a native speaker's own phrasing of the same query 6. To mitigate this, a random sample of the translated queries is checked by the researcher.

### 3.1.6. API call procedure
A Python script (Appendix B) dispatches each of the 1,000 queries, in each of the three languages, to each of the three model conditions, for a total of 9,000 calls. Calls are logged incrementally to disk as they complete, making the run resumable if interrupted rather than requiring a full restart. Each model is queried at a fixed delay between requests (3.0–4.0 seconds, set per model according to its published rate limit), and a failed call is retried up to three times with increasing backoff before being logged as an error and excluded from analysis. No system prompt is included at this stage: each query is submitted as bare user input, so that the recorded token count reflects the linguistic cost of the query text itself, independent of any prompt overhead 3.2.

### 3.1.7. Data retrieval
For every call, the script records: input, output, thinking, and cached token counts (as reported directly by the provider's own API response metadata); end-to-end latency; the raw response text; and, for failed calls, the error returned. Input and output token counts are the values each model's own tokenizer computed at inference time, making them exact rather than approximated figures.

### 3.1.8. Analysis method
The primary analysis is a two-way ANOVA on `input_tokens`, with Language and Model as fixed factors and the Language×Model interaction term reported explicitly alongside both main effects.JASP Version 0.96.0.0 is used as the analysis software. Standard ANOVA assumptions are checked on the residuals before F-statistics are reported: normality via Shapiro–Wilk and visual inspection of Q–Q plots, and homogeneity of variance via Levene's test. Token counts are non-negative and commonly right-skewed, so a log-transform of the dependent variable, or a non-parametric two-way alternative (the Scheirer–Ray–Hare test), is used as a fallback where these assumptions are materially violated; which approach was ultimately required is reported alongside the results in Chapter 4. Where a main effect or interaction is significant, Tukey HSD post-hoc comparisons identify which specific language or model pairs differ.

## 3.2. Design methodology (RQ2)
### 3.2.1. Hypotheses
RQ2 is tested as a family of hypotheses rather than a single H0, reflecting the three dependent variables the design intervention is evaluated against. For each of (i) processed token count, input tokens minus cached tokens, the fair cost comparison once Condition A's caching architecture is accounted for (Section 2.3), and (ii) response similarity between conditions, the null hypothesis is that Condition (User interface intervention, Condition A, versus raw-text prompting, Condition B) produces no significant difference, with Model and Language included as additional fixed factors alongside Condition. The primary design is therefore three-way: Condition × Model × Language, run separately for each dependent variable, with latency retained as a secondary, exploratory measure rather than a formally hypothesised outcome, since Section 2.3 already established that caching affects cost and latency through different mechanisms and need not move together.

### 3.2.2. Brainstorming session

Before any structured analysis of the MeQSum dataset, an initial ideation session was held at TU Delft's Faculty of Industrial Design Engineering to generate and narrow candidate directions for the interface intervention. Three participants took part, with backgrounds spanning computer science and industrial design, each assigned a distinct stakeholder role drawn directly from the cascade established in Figure 2.1: an end user of a multilingual health interface, a business provider building a product on top of a commercial LLM API, and a representative of the infrastructure layer concerned with compute cost and system efficiency. Structuring the session around these three roles was a deliberate methodological choice; it is the same systems-thinking lens carried through this report's Literature Review, applied here as a generative tool rather than only an analytic one. Verbal consent was obtained from all participants before the session began; no personally identifying information was recorded.

Participants generated candidate directions independently before sharing them with the group, a structural choice intended to avoid group discussion favouring whichever idea was most persuasively pitched rather than best on its own merits. Several directions were discussed and set aside at this stage, among them, a computer-vision-based gesture interface (rejected on privacy grounds) and a native-language-to-vector translation route (rejected because it relocates the token tax rather than removing it, since natural language generation would still be the user's input act). One direction consistently scored highest across all three stakeholder lenses: replacing free-text input with a visual interface the user directly manipulates to convey their intent, rather than typing it. This session produced a directional confidence to pursue visual, non-textual interaction as the intervention's core mechanism, and to set aside the alternatives, keeping the RQ2 in mind.

The first prototype built on this direction was a body-map interface, in which a user indicated symptom location directly on a visual avatar. Testing this prototype against the breadth of the MeQSum dataset exposed its limit quickly: a substantial share of consumer health queries, emotional or psychological concerns, medication- or caregiver-related questions, and administrative queries have no meaningful anatomical location to indicate on a body map at all. This gap is what motivated the systematic, dataset-driven analysis described next, rather than continuing to iterate on an anatomical metaphor that could not, by construction, cover the dataset's actual range of query types.

### 3.2.3. Design direction conceptualization

**Inductive coding procedure.** Rather than continuing to design against intuition, the full MeQSum corpus was coded systematically to determine what a structured interface would actually need to capture. For each of the 1,000 queries, the expert-authored *Summary*, rather than the longer, more digressive raw question text, was segmented into its constituent semantic clauses, following Ben Abacha and Demner-Fushman's (Ben Abacha & Demner-Fushman, 2019) own framing of their summaries as the essential informational content once peripheral narrative detail is stripped away. Each clause was then checked against a working checklist of candidate categories (an intent label, plus the parameter categories described below), and a *coverage* score was computed as the proportion of a query's clauses successfully mapped to at least one category. Queries reaching 80% coverage or higher were considered adequately representable by the emerging taxonomy; this threshold, rather than 100%, was chosen to tolerate the occasional idiosyncratic clause without letting rare edge cases dictate the whole category structure.

The coded corpus shows two things worth reporting directly. First, coverage is close to bimodal rather than smoothly distributed: queries reaching the 80% threshold average 99.96% coverage, while those falling short average only 21.1%. The taxonomy tends to capture a query almost completely or barely at all, which is a stronger validity signal for the category structure than a smooth, ambiguous middle ground would have been. Second, 524 of the 1,000 queries (52.4%) met the 80% threshold; a subsample of 275 of these was carried forward as the shortlist used in the design and testing phase (Section 3.2.5). Intent labelling across the coded corpus also surfaced a finding worth stating: of the six intent categories defined below, five occurred in the coded MeQSum sample (Information Seeking, Treatment Seeking, System Navigation, Symptom Checking, Emotional Support), while Clinical Triage did not occur at all, consistent with MeQSum's origin as questions submitted to a public information service rather than an urgent-care channel, and a boundary condition on the taxonomy's coverage claims returned to in Chapter 6.

**AI-assisted workflow.** Coding all 1,000 queries against the taxonomy checklist by hand was not feasible at this scale within the project's timeline; an LLM-assisted workflow was used instead with the help of Claude Sonnet 5 (Anthropic, 2026b), with the model prompted to segment each query's Summary into clauses and match each clause against the candidate category list, the same decomposition worked through by hand for a single illustrative query in Table 2.1, applied here at the scale of the full corpus, after which the researcher reviewed and adjudicated the output. This is disclosed here as a methodological choice with a direct consequence: the coding reflects a mixture of automated pattern-matching and human judgement rather than either alone, and any systematic bias in how the assisting model interprets ambiguous clauses is a limitation carried forward into the taxonomy's validity, addressed in Chapter 6.

**Parameter taxonomy.** The coding categories that emerged split into three types, each justified by a different measurement-theoretic argument developed in Section 2.5 (Table 3.1). Named clinical entities, a diagnosis, procedure, medication, or similar, are nominal (Stevens, 1946): open-class and unordered, so they are captured as an untagged, free-text *Keywords* field rather than forced into a fixed category set. Self-reported experiential intensities are ordinal and are captured as *FELT* axes, each scored 0–5, grounded partially in the affective/instrumental distinction from RIAS (Bales, 1950)(Section 2.5). Topical emphasis, how much of a query concerns a given theme, independent of anything felt, is also ordinal but structurally distinct, and is captured as *FOCUS* axes, each scored 0–3, structurally paralleling aspect-based sentiment's aspect-plus-attribute pattern (Pontiki et al., 2014). As stated in Section 2.5, this specific three-part split and the particular axes within it are a synthesis assembled for this design problem, not an instantiation of any single prior framework.

**Table 3.1:** The UI intervention parameter taxonomy

| Type | Scale | Categories / axes | Measurement basis |
|---|---|---|---|
| Keywords (nominal) | open, untagged | Diagnosis, Procedure, Treatment, Medications, Manufacturer, Demographics, Family/Caregiver, Comorbidity | Stevens (1946) |
| FELT axes (ordinal) | 0–5 | Symptom Severity (SYM), Emotional Distress (DIS), Duration (DUR) | RIAS / Bales (1950) |
| FOCUS axes (ordinal) | 0–3 | Pharmacology (PHR), Efficacy (EFF), Prognosis (PRG), Genetics (GEN), Diagnostic Method (DXM), Risk Factor (RSK), Referral (REF), Cost (CST), Clinical Trial (CTR) | (Pontiki et al., 2014) SemEval ABSA |

Six intent categories sit above these parameters, labelling what the user is fundamentally trying to accomplish rather than what the query contains: Information Seeking, Treatment Seeking, Symptom Checking, Clinical Triage, System Navigation, and Emotional Support.

**Language-agnostic encoding and caching strategy.** A query is submitted not as natural language but as a compact encoded string: a language tag, an intent code, any active FELT/FOCUS axis values, and the free-text keyword list, in a fixed format,

```
([LANG: <code>] [INTENT: <code>] [KIVIAT] <code>:<value> | ... [KEYWORDS] term1 | ...)
```

Every code, language, intent, and axis is a fixed English-language token regardless of the user's own language, so the only language-dependent content in the entire string is the free-text keyword list. This is a direct, structural answer to Section 2.2.1: because fragmentation severity depends on how much of a string a language-specific tokenizer must process, reducing the string to mostly fixed, English-coded tokens reduces exposure to the tax by construction, rather than by compressing an existing natural-language sentence after the fact (Section 2.3).

This fixed-format string is paired with an intentionally long system prompt (a decode dictionary explaining every code, plus per-intent response guidance) (Appendix E) engineered to exceed the minimum length commercial providers require before context caching activates. Once cached, this prompt's cost is paid in full only once per cache lifetime rather than on every call, meaning the token cost of a Condition A call is dominated by the short, freshly submitted encoded string, not by the instructions needed to interpret it, the direct application of the caching mechanism reviewed in Section 2.3. Chapter 4 reports the extent to which this caching benefit actually materialised, and to what degree it varied by provider.

The interface translating a user's interaction into this encoded string, the wireframing and prototyping process itself, was likewise developed through an AI-assisted workflow, detailed in Section 3.2.4.

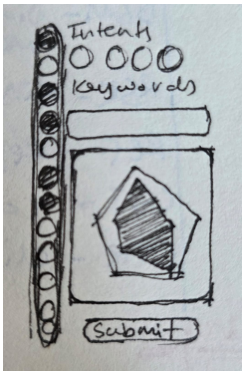
### 3.2.4. Wireframing logic and iterations

**Why a Kiviat interface.** The FELT and FOCUS axes established in Section 3.2.3 are, by design, simultaneous and multi-dimensional: a single query can carry a meaningful value on several axes at once, and what the interface needs to convey, and let the user set, is a profile across all of them together, not one value in isolation. A Kiviat diagram (radar chart) is built exactly for this case: each axis is rendered as a spoke radiating from a shared centre, and the enclosed shape formed by connecting a value on every spoke gives an immediate visual summary of a multi-attribute profile that a set of separate sliders or a numbered list would not (Thaker et al., 2016). Radar charts have precedent specifically in healthcare communication: Thaker et al. (Thaker et al., 2016) used a radar chart to visualise multiple patient-reported outcome domains, sexual function, urinary and bowel symptoms, vitality, alongside cost, for prostate cancer patients comparing treatment options, on the same structural logic MedQuery's FELT/FOCUS axes rely on: several self-reported or outcome dimensions, read together as one shape rather than as isolated numbers.

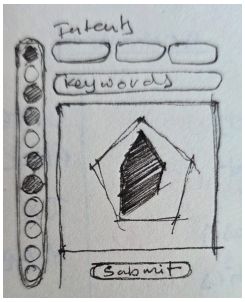
This precedent needs an honest complication rather than a selective reading of it. Turchioe et al. (2019) systematic review of patient-facing health data visualisations found that radar charts appeared in very few of the reviewed studies, well behind simpler formats such as line graphs, number lines, and bar graphs, and that these simpler formats were generally better understood by patients in the studies that measured comprehension directly. A Kiviat interface is therefore not the safe, low-effort default in patient-facing design; it is a deliberate trade-off, justified here specifically because the FELT/FOCUS structure is genuinely multivariate and simultaneous in a way a single bar or number line cannot represent at all, which is precisely the condition under which a radar chart's advantage over simpler formats is greatest rather than merely stylistic.

One further distinction is worth making explicit, since it separates this design from its precedent rather than simply borrowing it. Thaker et al. (2016)'s radar charts, and the wider literature it sits within, are *output* displays: a static visualisation of data already collected. The intervened User interface with Kiviat is instead an *input* control; adjusting a spoke's vertex is the act of specifying a FELT or FOCUS value, following the direct-manipulation logic established in Section 2.6 (Shneiderman, 1983), rather than a chart rendered after the fact. The visual grammar is the same; what this thesis repurposes it for is not.

**Iteration history.** The Kiviat interface was the fourth form the design took. The initial prototype used calendar-style dropdowns to capture symptom timing, but proved unable to represent anything beyond surface-level scheduling detail; it could not translate a query's deeper experiential content at all. The second iteration replaced this with a flexible, node- and gesture-based interaction closer to a check-in tool, capturing valence, arousal, intensity, and trajectory through direct touch gestures rather than form fields; this captured more nuance, but required a separate gesture for every dimension, producing too many discrete interactions for what should have been a single, unified act of query formulation. The third iteration addressed this by binding every active dimension to a single combined diagram the user manipulates as one object rather than several. The fourth and current iteration, shown in Figure 3.2, refined this into its present sequence: select an intent, enter free-text keywords, then select and weight the relevant FELT/FOCUS parameters directly on the Kiviat diagram, establishing both which axes are active and their relative values in a single interaction. The full sketch sequence across all four iterations is provided in Appendix D.



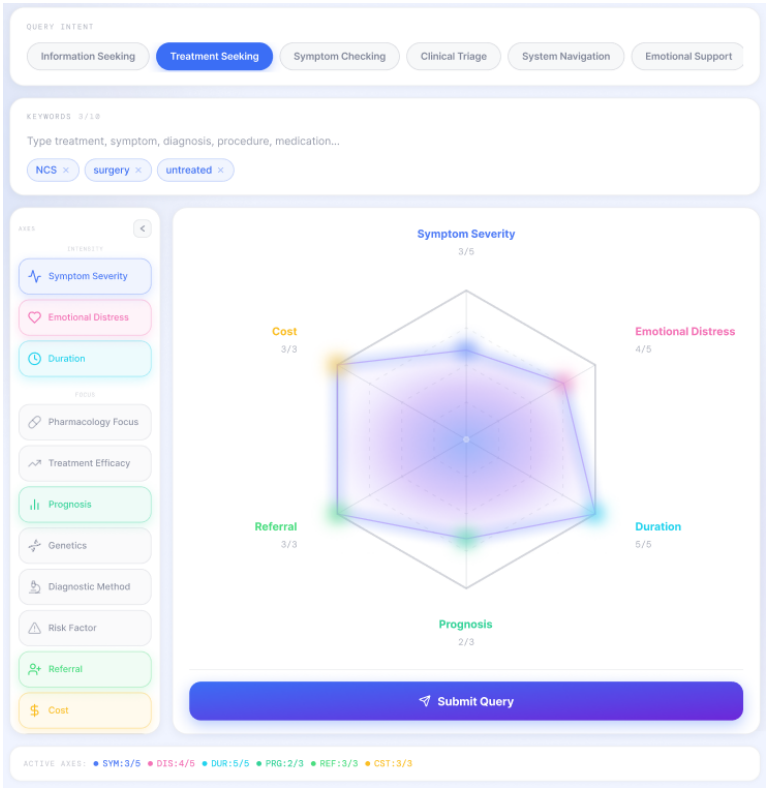

**(a)** Example 1 - Low-fidelity concept for UI wireframe.




**(b)** Example 2 - Low-fidelity concept for UI wireframe.

**Figure 3.1:** Concept Visualization for fourth UI iteration

To make this concrete, Figure 3.2 shows the same query decomposed by hand in Table 2.1 (Section 2.5), a ten-year, untreated condition, refused surgery on cost grounds, and a plea for help, as it appears in the high-fidelity MedQuery interface, alongside the encoded string that the interaction actually produces. The visual grammar of the Kiviat diagram serves a dual purpose: it lets a user simultaneously manipulate multiple experiential axes at once (Figure 3.2, left), while the interface systematically translates that same rich intent into a highly compressed, language-agnostic string (Figure 3.2, right), the object actually submitted to the model. The six active vertices, Symptom Severity (3/5), Emotional Distress (4/5), Duration (5/5), Cost (3/3), Referral (3/3), and Prognosis (2/3) , are exactly the FELT and FOCUS values inferred from the query's text in Table 2.1, and the three keywords (NCS, surgery, untreated) are the same nominal entities identified there.




(a) High-fidelity interface: intent, keywords, and the Kiviat diagram for the worked example in English.

```
[LANG: EN]
[INTENT:
Treatment_Seek]
[KIVIAT]
SYM:3 | DIS:4 | DUR:5 |
PRG:2 | REF:3 | CST:3
[KEYWORDS]
NCS | surgery |
untreated
```

(b) The resulting encoded string, submitted to the model in place of the original free-text query.

**Figure 3.2:** The same worked example (Table 2.1) as manipulated in the MedQuery interface (left) and the compact, language-agnostic encoded string it produces (right).

Because the interface's codes, language, intent, and axis labels are fixed English-language tokens regardless of the user's own language (Section 3.2.3), the same worked example produces a structurally identical encoded string whether a user interacts with the Hindi or Arabic localisation of the interface or the English one shown in Figure 3.2; only the `[LANG:]` tag and the free-text `[KEYWORDS]` content change. Screenshots of the same query in all three interface languages are provided in Appendix D. The user interface prototype can be accessed through the QR code in Appendix F.

### 3.2.5. Testing Condition A vs Condition B

Figure 3.3 summarises the procedure detailed in this section: the two conditions diverge only in how a query is formed and what system prompt accompanies it, then converge onto an identical logging and analysis pipeline, so that any difference detected between them can be attributed to that divergence rather than to any difference in how the two conditions were measured.

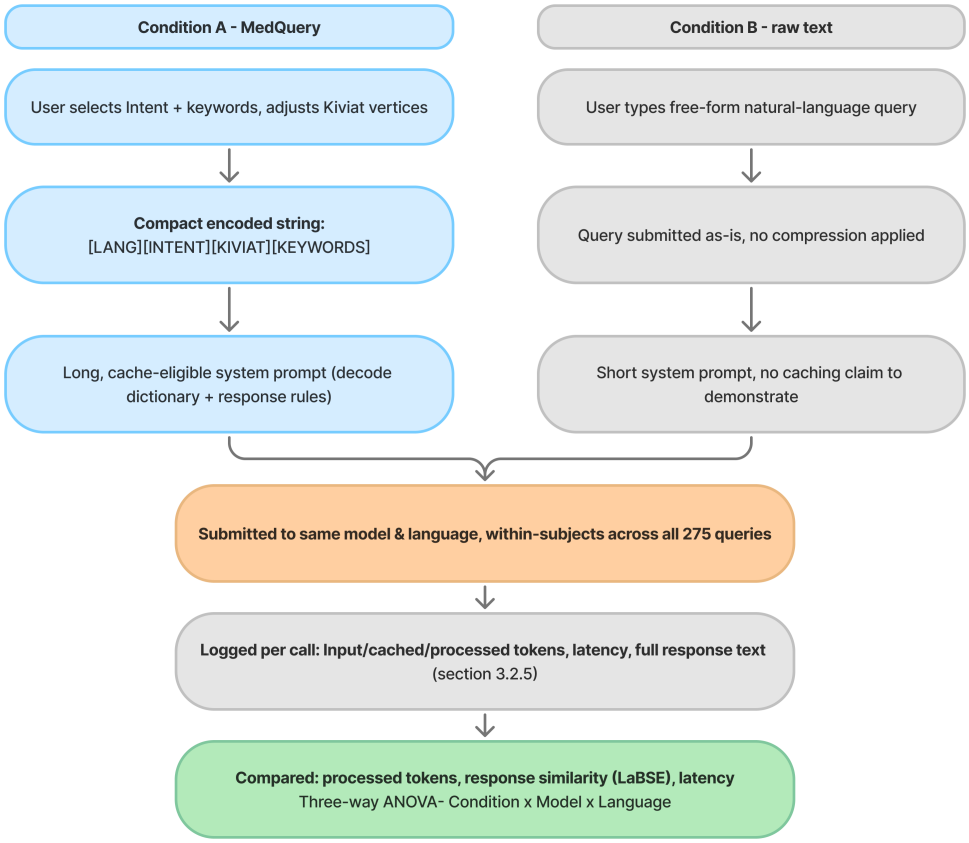



**Figure 3.3:** Testing procedure for Condition A (MedQuery) versus Condition B (raw text), from query formation through to statistical comparison.

**Procedure.** Each of the 275 shortlisted queries (Section 3.2.3) is run under both conditions(Appendix C), across all three languages and three models, following the same call structure established in Section 3.1.6: $275 \times 3 \times 3 \times 2 = 4{,}950$ calls in total. Condition A submits the query as a UI-encoded string against the long, cache-eligible system prompt described in Section 3.2.3; Condition B submits the same query as free-form natural-language text against a short, deliberately unexpanded system prompt with no caching claim to demonstrate. For every call, input tokens, cached tokens, latency, and the full response text are logged, mirroring the fields collected in Section 3.1.7. Cached tokens allow `processed_tokens` (input tokens minus cached tokens) to be computed post hoc as the fair cost

comparison between conditions, since raw input tokens alone would unfairly penalise Condition A for tokens it does not pay full price for once cached (Section 3.2.3).

**Response similarity.** The stored response text from both conditions is embedded using LaBSE, a language-agnostic sentence embedding model chosen specifically for its validated cross-lingual performance across English, Hindi, and Arabic and its independence from any single model provider. Cosine similarity between a query's Condition A and Condition B response embeddings gives a single similarity score per query, model, and language cell. This measure is referred to throughout as *response similarity* rather than "accuracy," since that is what it actually measures: agreement between what the two conditions produced, not correctness against any independent ground truth. Two failure modes follow directly from this and are carried forward to Chapter 5 rather than treated as resolved here: both conditions could converge on the same incorrect or generically hedged answer and still score highly similar; and both could be individually reasonable while scoring as dissimilar if they differ in emphasis rather than in correctness, a real possibility here specifically, since Condition A's FOCUS axes are designed to bias which topics a response emphasises, a deliberate difference in content allocation rather than an error.

**Analysis method.** Similar to subsection 3.1.8, JASP Version 0.96.0.0 is used as the analysis software.The primary analysis is a three-way ANOVA, Condition $\times$ Model $\times$ Language, run separately for each of the two hypothesised dependent variables named in Section 3.2.1:
`processed_tokens` and response similarity. The same assumption-checking procedure specified in Section 3.1.8 is applied to both, Shapiro–Wilk and Q–Q inspection for normality, Levene's test for homogeneity of variance, with a log-transform or non-parametric alternative as a fallback where assumptions are materially violated, and Tukey HSD post-hoc comparisons are reported wherever a main effect or interaction is significant, with particular attention to the Condition$\times$Model interaction term, since Section 2.3 already establishes a mechanistic reason (provider-specific caching behaviour) why the token-cost benefit of Condition A might not be uniform across models. Latency is analysed descriptively alongside these two primary dependent variables rather than as a third formally hypothesised outcome, as outlined in Section 3.2.1.

# 4

# Results

## 4.1. RQ1: The token tax

**Assumption checks.** A two-way ANOVA was run on `input_tokens`, with Language (English, Hindi, Arabic) and Model (`gemini_25_flash_lite`, `gemini_31_flash_lite`, `gpt54`) as fixed factors, across 8,812 successful calls. Levene's test indicated significant heterogeneity of variance across the nine language×model cells ($F(8, 8803) = 15.52$, $p < .001$), consistent with token counts being non-negative, right-skewed count data rather than a normally distributed continuous measure. Given the large sample size and the resulting robustness of the F-test under the central limit theorem, results are reported directly, with Games–Howell post-hoc comparisons used in place of Tukey HSD wherever the two diverge, per the contingency specified in Section 3.1.8.

**Main effects.** Both main effects are statistically significant. Language has a significant effect on input token count, $F(2, 8803) = 95.85$, $p < .001$, $\eta_p^2 = 0.021$; Model also has a significant, smaller effect, $F(2, 8803) = 7.55$, $p < .001$, $\eta_p^2 = 0.008$. The Language×Model interaction is significant but small, $F(4, 8803) = 6.77$, $p < .001$, $\eta_p^2 = 0.003$. H0, that input token count does not differ by language or model, is rejected: a token tax is present in this dataset (Table 4.1, Figure 4.1).

**Table 4.1:** Two-way ANOVA on `input_tokens` (RQ1 benchmark), Type III sums of squares

| Source | SS | df | MS | F | p | $\eta_p^2$ | $\omega^2$ |
|---|---|---|---|---|---|---|---|
| Language | 985,500 | 2 | 492,700 | 95.85 | $< .001$ | 0.021 | 0.021 |
| Model | 180,400 | 2 | 90,210 | 7.55 | $< .001$ | 0.008 | 0.004 |
| Language × Model | 139,100 | 4 | 34,780 | 6.77 | $< .001$ | 0.003 | 0.003 |
| Residuals | $4.53 \times 10^7$ | 8803 | 5141 | | | | |

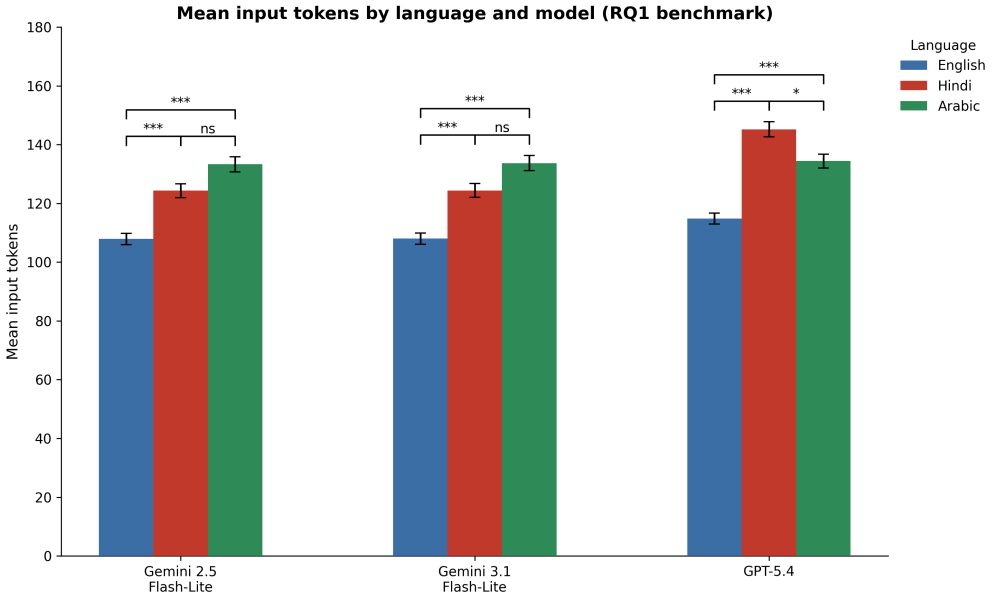



**Figure 4.1:** Mean input tokens by language and model, RQ1 benchmark (error bars: ±1 SE). English is consistently cheapest; Hindi and Arabic are consistently more expensive, with the gap widest on GPT-5.4.

**Post-hoc comparisons.** Games–Howell comparisons on Language show English differs significantly from both Hindi and Arabic (English–Arabic: $MD = -23.54$, $SE = 1.811$, $p < .001$; English–Hindi: comparable magnitude and direction, $p < .001$), while Hindi and Arabic do not differ significantly from each other ($MD = 2.36$, $SE = 2.02$, $t(5371) = 1.17$, $p = .472$). This is a more specific finding than a blanket "non-English is more expensive" result: the tax separates English from both other languages cleanly, but does not, in this dataset, separate Hindi from Arabic; both carry a statistically indistinguishable penalty relative to English. On Model, `gpt54` differs significantly from both Gemini variants (both $p < .001$), while `gemini_25_flash_lite` and `gemini_31_flash_lite` do not differ significantly from each other ($MD = 0.22$, $p = .993$), consistent with the two Gemini generations sharing tokenizer behaviour, as anticipated in Section 3.1.3.

**Latency.** As a secondary, exploratory measure, a parallel two-way ANOVA was run on `latency_ms`. Language's effect on latency is not significant, $F(2, 8803) = 2.83$, $p = .059$, $\eta_p^2 < .001$ , language moves token count but not response time. Model's effect on latency is large and significant, $F(2, 8803) = 1025$, $p < .001$, $\eta_p^2 = 0.189$: `gemini_31_flash_lite` is both markedly slower (means in the 5,500–6,100ms range) and far more variable (coefficients of variation of 1.5–1.9, versus 0.2–0.6 for the other two models) than either `gemini_25_flash_lite` (1,600–1,730ms) or `gpt54` (7,650–9,165ms, slower on average but far more consistent). The Language×Model interaction on latency is significant but small, $F(4, 8803) = 8.50$, $p < .001$, $\eta_p^2 = 0.004$. Read together with the token result, this is an early instance of a pattern this thesis returns to repeatedly: token count and latency are governed by different mechanisms and do not move together, a point developed further in Section 4.2 and discussed in Chapter 5.

## 4.2. RQ2: Does the Medquery reduce the token tax?

**Overview.** A three-way ANOVA (Condition × Model × Language) was run on `processed_tokens` across 4,941 valid trials. All three main effects and the Condition×Model and Condition×Language two-way interactions are significant; the three-way interaction is not, $F(4, 4932) = 0.87$, $p = .481$. The pattern reported below is stable across all three languages, not an artefact confined to one of them (Table 4.2).

**Table 4.2:** Three-way ANOVA on `processed_tokens` (RQ2, Condition × Model × Language), Type III sums of squares

| Source | SS | df | MS | F | p | $\eta_p^2$ | $\omega^2$ |
|---|---|---|---|---|---|---|---|
| Condition | $3.88 \times 10^5$ | 1 | $3.88 \times 10^5$ | 501.2 | $< .001$ | 0.092 | 0.031 |
| Model | $4.34 \times 10^7$ | 2 | $2.17 \times 10^7$ | 280.0 | $< .001$ | 0.532 | 0.346 |
| Language | 202,200 | 2 | 101,100 | 13.04 | $< .001$ | 0.005 | 0.001 |
| Condition × Model | $3.95 \times 10^7$ | 2 | $1.97 \times 10^7$ | **254.5** | $< .001$ | **0.508** | 0.314 |
| Condition × Language | 114,700 | 2 | 57,330 | 7.40 | $< .001$ | 0.003 | $< .001$ |
| Model × Language | 173,200 | 4 | 43,300 | 5.59 | $< .001$ | 0.005 | 0.001 |
| Condition × Model × Language | 26,950 | 4 | 6,737 | 0.87 | .481 | $< .001$ | 0.000 |
| Residuals | $3.82 \times 10^7$ | 4932 | 7749 | | | | |

**Main result.** The Condition×Model interaction is large, $F(2, 4932) = 254.5$, $p < .001$, $\eta_p^2 = 0.508$, by a wide margin the largest effect in this analysis, and larger than either main effect. Figure 4.2 shows why: the direction of the token-cost effect flips depending on which model is used. On both Gemini variants, MedQuery (Condition A) is cheaper than raw-text prompting (Condition B) by 70.23 processed tokens on average ($SE = 4.33$, $t(4932) = -16.20$, $p < .001$). On GPT-5.4, the same comparison reverses: Condition A is *more* expensive than Condition B by 308.5 processed tokens ($SE = 4.33$, $t(4932) = 71.18$, $p < .001$).

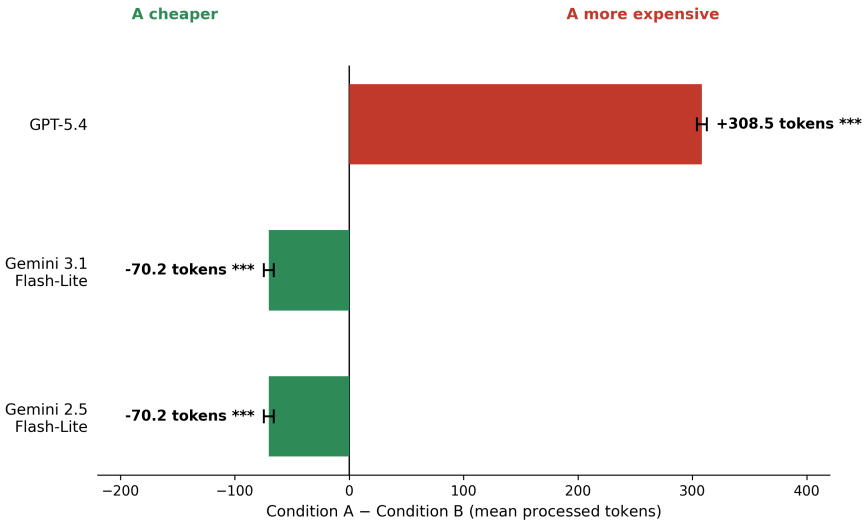



**Figure 4.2:** Condition A − Condition B, mean processed tokens, by model (Tukey HSD, averaged over language). The Gemini variants are identical (shared tokenizer); GPT-5.4 reverses the effect entirely.

**Insight** This reversal is not noise; it follows directly from the caching-architecture difference established in Section 2.3. Descriptively, Condition A's processed-token cost on the Gemini models sits in the 43–49 token range across all three languages, close to the short, freshly submitted encoded string alone, consistent with Gemini's explicit caching absorbing nearly all of the long system prompt. On GPT-5.4, Condition A's processed-token cost is an order of magnitude higher, 415–456 tokens depending on language, and its coefficient of variation on Arabic specifically is striking: 0.025, versus 0.43–0.57 on English and Hindi, a level of consistency that suggests GPT-5.4's automatic caching is hitting a stable but partial cutoff on Arabic input rather than caching the prompt as completely as Gemini does. The practical consequence is that GPT-5.4's caching does not absorb enough of the long system prompt to make Condition A's fixed overhead cheaper than Condition B's short, uncached one, the reverse of what the architecture was designed to achieve for that provider.

**Reframing the claim.** This result does not support the original, unconditional hypothesis that a structured, visual-first interface reduces token cost. Rather, it directs a narrower and precise insight: the

Medquery reduces token cost *contingent on the underlying provider's caching mechanism fully absorbing the structured prompt*. That condition held for both Gemini variants tested and did not hold for GPT-5.4. This distinction is carried forward as the central claim of this thesis and is returned to directly in Chapter 5.

**Other effects.** The Model main effect is also large, $F(2, 4932) = 280.0$, $p < .001$, $\eta_p^2 = 0.532$, driven by the same GPT-5.4 asymmetry: Games–Howell comparisons show the two Gemini variants do not differ from each other ($MD = 0.00, p = 1.000$) while GPT-5.4 differs from both by roughly 199 tokens ($p < .001$ in both cases). The Condition main effect is significant but comparatively modest, $F(1, 4932) = 501.2$, $p < .001$, $\eta_p^2 = 0.092$ (Games–Howell: $MD = 86.0$ tokens, $p < .001$, Condition A more expensive *on average* once GPT-5.4's cost is folded in), a reminder that an unweighted "average effect of Condition" would itself be a misleading headline, since it obscures the fact that the sign of the effect depends entirely on which model is queried. The Language main effect is significant but small, $F(2, 4932) = 13.04$, $p < .001$, $\eta_p^2 = 0.005$; pairwise comparisons are mostly non-significant (Games–Howell: ar–en $p = .112$, ar–hi $p = .617$), with only en–hi reaching significance at a small magnitude ($MD = -15.4$, $p = .028$), the token tax detected in RQ1 (Section 4.1) survives into Condition A's encoded-string format only weakly, consistent with the design intent that free-text keywords are the only remaining language-dependent content in the string (Section 3.2.3).

## 4.3. Response similarity

A three-way ANOVA on response similarity (Model $\times$ Language; Condition is not a factor here, since similarity is inherently a cross-condition measure computed per query) was run across 4,941 valid pairs. The Model main effect is not significant, $F(2, 4935) = 2.58$, $p = .078$, $\eta_p^2 = 0.001$. Language and the Model$\times$Language interaction are both statistically significant, $F(2, 4935) = 22.23$, $p < .001$, $\eta_p^2 = 0.009$; $F(4, 4935) = 11.21$, $p < .001$, $\eta_p^2 = 0.009$, but both effect sizes are negligible in practical terms, each accounting for under 1% of variance despite reaching significance at this sample size. Descriptively, mean similarity scores cluster tightly in a narrow band across every model$\times$language cell (0.576–0.625; Table 4.3), with no cell standing out as an outlier and no consistent pattern favouring any one language.

**Table 4.3:** Mean response similarity by model and language

| Model | Arabic | English | Hindi |
|---|---|---|---|
| Gemini 2.5 Flash-Lite | 0.585 | 0.613 | 0.613 |
| Gemini 3.1 Flash-Lite | 0.579 | 0.581 | 0.625 |
| GPT-5.4 | 0.603 | 0.576 | 0.603 |

This is a clean result, but its interpretation depends on what response similarity actually measures. As established in Section 3.2.5, a similarity score in this range reflects moderate agreement between conditions, not correctness against any ground truth, and the two failure modes flagged there, shared convergence on a generic or incorrect answer, and legitimate divergence driven by Condition A's FOCUS-axis emphasis rather than any error, remain live interpretive possibilities that this measure alone cannot distinguish. That interpretation is taken up in Chapter 5.

## 4.4. Latency

Latency was analysed as a secondary, exploratory measure via a three-way ANOVA (Condition $\times$ Model $\times$ Language), summarised in Table 4.4. Unlike the clean processed-tokens result, every effect in this analysis is significant, including the three-way interaction, $F(4, 4932) = 87.42$, $p < .001$, $\eta_p^2 = 0.052$. Latency's pattern is not stable across languages the way the token result is, a genuine complication rather than an artefact to smooth over.

**Table 4.4:** Three-way ANOVA on `latency_ms` (RQ2), Type III sums of squares

| Source | SS | df | MS | F | p | $\eta_p^2$ | $\omega^2$ |
|---|---|---|---|---|---|---|---|
| Condition | $7.57 \times 10^8$ | 1 | $7.57 \times 10^8$ | 336.4 | $< .001$ | 0.064 | 0.014 |
| Model | $4.66 \times 10^{10}$ | 2 | $2.33 \times 10^{10}$ | 10,360 | $< .001$ | **0.808** | 0.652 |
| Language | $4.30 \times 10^9$ | 2 | $2.15 \times 10^9$ | 956.5 | $< .001$ | 0.279 | 0.060 |
| Condition × Model | $1.59 \times 10^9$ | 2 | $7.99 \times 10^8$ | 353.0 | $< .001$ | 0.125 | 0.022 |
| Condition × Language | $4.76 \times 10^8$ | 2 | $2.39 \times 10^8$ | 106.5 | $< .001$ | 0.041 | 0.007 |
| Model × Language | $6.06 \times 10^9$ | 4 | $1.52 \times 10^9$ | 676.7 | $< .001$ | 0.384 | 0.085 |
| Condition × Model × Language | $6.06 \times 10^8$ | 4 | $1.51 \times 10^8$ | 87.42 | $< .001$ | 0.052 | 0.008 |
| Residuals | $1.11 \times 10^{10}$ | 4932 | $2.25 \times 10^6$ | | | | |

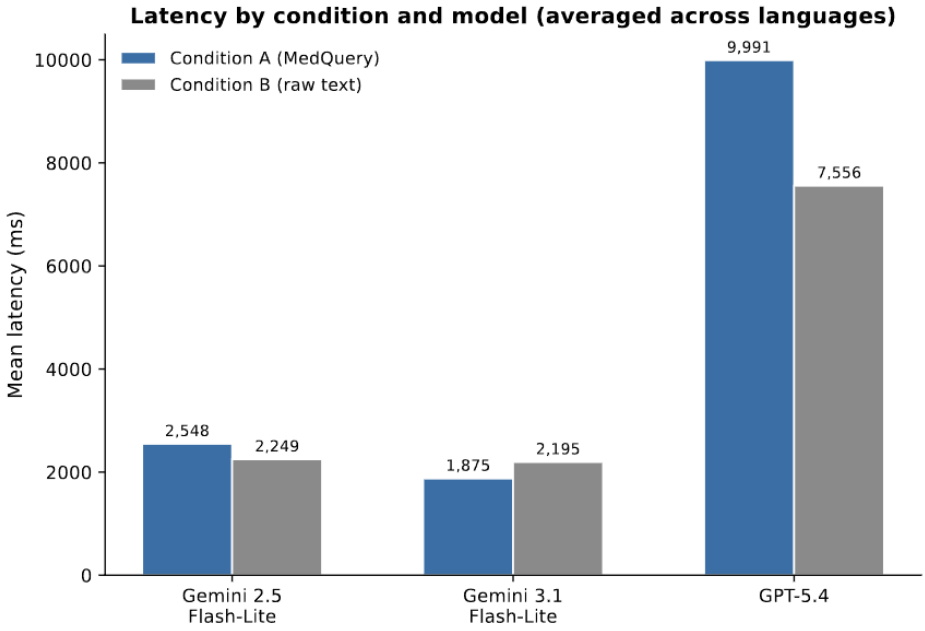



**Figure 4.3:** Mean latency by condition and model, averaged across languages. Model dominates latency ($\eta_p^2 = 0.808$) far more than Condition does.

Model accounts for the overwhelming share of variance in latency ($\eta_p^2 = 0.808$), an order of magnitude larger than its effect on token cost (Table 4.2), and Figure 4.3 shows why: GPT-5.4 is slower than either Gemini variant by a wide margin regardless of condition, averaging roughly 10 seconds per call versus 1.9–2.5 seconds for the Gemini models. Two cell-level results illustrate that token savings and latency savings are governed by separate mechanisms, exactly as anticipated in Section 2.3. On Gemini 2.5 Flash-Lite for Hindi, Condition A is cheaper in tokens (Section 4.2) yet slower in latency: 2,953ms versus 2,368ms, roughly 25% slower despite costing fewer tokens. On GPT-5.4 for Hindi, Condition A is slower still, 13,540ms versus 9,399ms, approximately 44% slower, but here the latency cost moves in the same direction as the token cost rather than against it, consistent with the same caching shortfall responsible for both (Section 4.2). Caching reduces cost; it does not guarantee that it reduces, or even leaves unchanged, response time, and the two dependent variables should not be assumed to track each other in either direction.

## 4.5. Summary

Table 4.5 consolidates every dependent variable reported in this chapter, averaged across the three languages per model, as a single point of reference.

**Table 4.5:** Consolidated results summary by model and language

| Model | Lang. | RQ1 input tok. | RQ2: processed tokens Cond. A | Cond. B | A − B | Response similarity | Latency (ms) Cond. A | Cond. B |
|---|---|---|---|---|---|---|---|---|
| Gemini 2.5 Flash-Lite | Arabic | 133.3 | 48.7 | 127.2 | $-78.5$ | 0.585 | 2,632 | 2,348 |
| | English | 107.5 | 43.2 | 103.5 | $-60.3$ | 0.613 | 2,059 | 2,030 |
| | Hindi | 124.3 | 47.0 | 118.8 | $-71.8$ | 0.613 | 2,953 | 2,368 |
| Gemini 3.1 Flash-Lite | Arabic | 133.7 | 48.7 | 127.2 | $-78.5$ | 0.579 | 1,989 | 2,204 |
| | English | 108.0 | 43.2 | 103.5 | $-60.3$ | 0.581 | 1,757 | 2,198 |
| | Hindi | 124.4 | 47.0 | 118.8 | $-71.8$ | 0.625 | 1,878 | 2,182 |
| GPT-5.4 | Arabic | 134.4 | 415.5 | 128.4 | $+287.1$ | 0.603 | 10,850 | 8,002 |
| | English | 114.8 | 430.8 | 110.1 | $+320.7$ | 0.576 | 5,583 | 5,267 |
| | Hindi | 145.2 | 456.4 | 138.7 | $+317.7$ | 0.603 | 13,540 | 9,399 |

Four patterns are visible at a glance. First, the RQ1 token tax (input tokens) is present within every model, and consistently orders Arabic $\approx$ Hindi $>$ English within each model row, the tax exists at the raw-query level regardless of which model eventually processes it, and RQ1's finding that Arabic and Hindi are statistically indistinguishable (Section 4.1) is visible here as their consistently close values. Second, the Condition A$-$B column makes the central reversal of this thesis immediately legible: negative for every Gemini cell, positive for every GPT-5.4 cell, and stable in direction across all three languages within each model, the model-level pattern established in Section 4.2, confirmed here at the language level. Third, response similarity is essentially flat both across models and across languages within each model (0.576–0.625), with no cell standing out. Fourth, latency does not track either token column: GPT-5.4 is an order of magnitude slower than either Gemini variant in every language and condition, and within the Gemini rows, Condition A is sometimes slower despite being cheaper (e.g., both Gemini variants on Hindi). No single dependent variable tells the full story in isolation; it is the pattern across all four, read together, that motivates the discussion in Chapter 5.

# 5

# Discussion

This chapter has four moves. Section 5.1 reframes the results in a more precise, more mechanistically grounded, and more useful extension of the hypothesis as posed originally. Section 5.2 returns explicitly to the systems-thinking lens introduced in Section 1.1, locating this finding within the stakeholder cascade and layered architecture established there. Section 5.3 and Section 5.4 take up the two secondary results, response similarity and latency, and what their caveats mean with reference to the thesis's claims. Section 5.5 closes on what a designer, not just a researcher, should take from this work.

## 5.1. Reframing the central claim

As originally posed, RQ2 asked a binary question: does a visual-first, structured-query interface reduce token cost relative to raw-text prompting? Framed this way, the honest answer is unsatisfying: *sometimes, and sometimes emphatically not.* On both Gemini variants tested, the Medquery reduced processed-token cost by roughly 70 tokens per query. On GPT-5.4, it increased cost by roughly 309 tokens per query, more than four times the size of the saving it produced elsewhere, in the opposite direction.

However, the most accurate description of what Section 4.2 found. The Condition$\times$Model interaction was the single largest effect in the entire RQ2 analysis ($\eta_p^2 = 0.508$, larger than either main effect, larger than every other interaction reported in this thesis) and the three-way interaction with Language was not significant, meaning this is not a noisy, marginal, or language-dependent result. It is a large, stable, well-characterised effect, and its content is not "the intervention works" or "the intervention fails," but a specific, mechanistic boundary condition: *Medquery reduces token cost when, and only when, the model provider's caching mechanism absorbs the structured system prompt close to completely.* That condition held for Gemini's explicit caching architecture, which the descriptive data in Section 4.2 show absorbing the vast majority of the prompt's cost regardless of language. It did not hold for GPT-5.4's automatic caching, which left a remainder large enough, roughly 400 tokens after caching, on the descriptive evidence, to exceed Condition B's entire uncached prompt on its own.

An unconditional claim, "visual-first interfaces reduce token cost", would have been falsified by a third of this thesis's own data and would have offered no account of why. The conditional claim survives all of the data. It explains the direction of the effect in every cell rather than describing it, and converts what would otherwise look like a contradiction into a testable, provider-specific prediction. It suggests that before deploying an interface of this kind, check whether the target provider's caching architecture absorbs a long, structured system prompt close to completely, because the intervention's sign depends on that answer more than on anything about the interface design itself. Donella Meadows's framing of leverage points, quoted at the opening of this report, is apt here in a way that only becomes visible after running the numbers: a leverage point is not powerful because of what it is in isolation, but because of how the rest of the system is configured to receive it. The interface layer is real leverage; Section 4.2's Gemini result demonstrates that directly, but it is leverage exercised *through* the caching layer imme-

diately downstream of it, not independently of it. Section 5.2 develops this point explicitly against the stakeholder cascade introduced in Section 2.1.

## 5.2. Back to the systems lens

Section 2.1 mapped the path an AI health query travels through five layers: model developers, the API/business layer, application/UI design, end users, and downstream equity effects (Figure 2.1), and argued that problems visible at the interface are frequently symptoms of decisions made further upstream. The GPT-5.4 result is a direct empirical instance of that argument.

Consider what actually changed, and what did not, between the Gemini and GPT-5.4 conditions. The interface layer did not change: The Medquery produced the same kind of compact, mostly language-independent encoded string, wrapped in the same long, deliberately cache-eligible system prompt, regardless of which model received it (Section 3.2.3). What changed sits one layer upstream, at the API/business layer of Figure F.1: Gemini's explicit caching absorbed the system prompt close to completely; GPT-5.4's automatic caching did not.

The reversal observed on GPT-5.4 is not an anomaly; it is a direct consequence of opaque, provider-specific caching mechanics. As mapped in Figure D.5, while Gemini's explicit architecture fully absorbed the structured prompt, GPT-5.4's automatic caching hit a premature cutoff, forcing the user to pay for the system's overhead on every call. This visually underscores why interface leverage cannot be evaluated in isolation from the business layer beneath it.

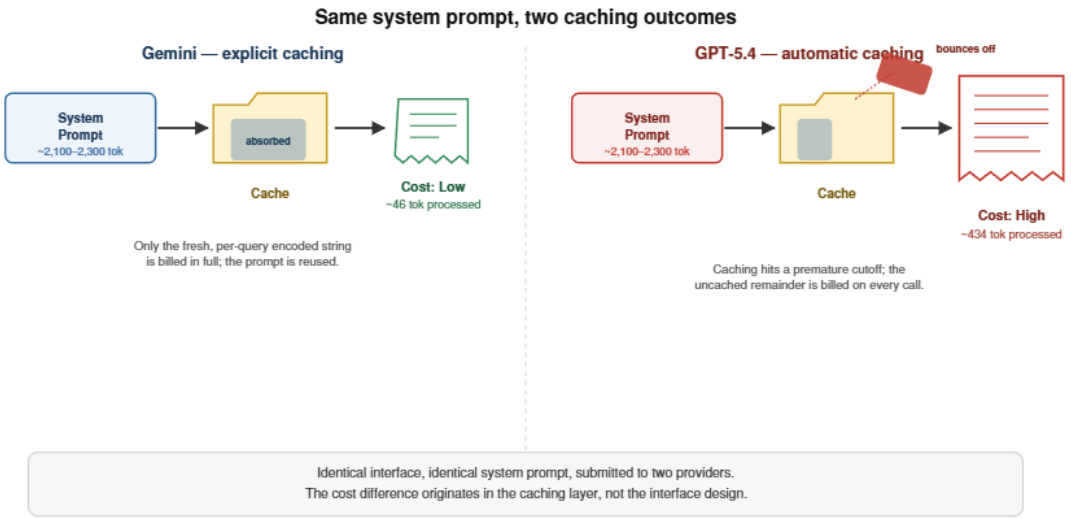



**Figure 5.1:** The same system prompt, submitted to two providers, meets two different caching outcomes. The cost difference originates one layer above the interface, not within it.

The token-cost reversal reported in Section 4.2 is therefore not a failure of the interface design being evaluated; it is a failure, or at least an inconsistency, at the caching layer, surfacing as a symptom the interface layer cannot see or correct on its own. An interface designer inspecting only the UI layer, with no visibility into provider-side caching behaviour, would have no way to predict this reversal from the design itself; it only becomes visible once the full stack is traced, which is precisely the diagnostic habit the systems-thinking lens (Section 1.1) was adopted to encourage.

This matters for how the scope boundary drawn in Section 2.1 should be read. That boundary excluded the API/business and model layers on practical grounds; they are not accessible to an independent interface designer. The RQ2 result shows that exclusion has a cost as well as a justification: an intervention correctly designed and correctly scoped to the one layer available to it can still be overturned by a layer its designer cannot touch. This is a different failure mode from the one RQ1 characterises. RQ1's token tax originates at the model and tokenizer layer, calibrated once at training time and effectively fixed thereafter (Section 2.2.1); it is stable, predictable, and, as Section 4.1 confirms, consistent in direction

across every model tested. The caching layer responsible for RQ2's reversal is neither stable nor predictable in the same way: it is a business-layer implementation choice that two otherwise-comparable providers made differently, with no external signal available to the interface designer indicating which choice a given provider has made. A structural tax, calibrated once and holding steady, is a target a designer can at least reason about and plan around, as this thesis's Condition A does. An inconsistent, opaque, provider-specific business-layer mechanism is not; it can only be measured empirically, provider by provider, which is exactly what the benchmarking approach in this thesis was forced to do in order to detect the reversal at all.

Section 5.1 already showed that the practical implications for an interface-level intervention are possible and it is real leverage where the downstream condition holds. Furthermore, the leverage exercised at one layer of a multi-layer system cannot be evaluated, or trusted, by inspecting that layer alone. Section 5.5 returns to what this means concretely for anyone following the method and designing a similar interface in practice.

## 5.3. Response similarity and the FOCUS-axis caveat

Section 4.3 reported response similarity scores clustered tightly between 0.576 and 0.625 across every model and language, with Model contributing no significant effect and Language contributing a statistically significant but practically negligible one. Read alone, a moderate similarity score in this range is genuinely ambiguous: Section 3.2.5 flagged two failure modes that could produce it for opposite reasons, both conditions converging on the same generic or incorrect answer, which would inflate similarity without indicating quality, or both conditions producing individually reasonable but differently emphasised responses, which would suppress similarity without indicating error. The measure cannot distinguish these on its own.

The pattern of the result, rather than its magnitude alone, offers a reason to favour the second reading. If the moderate similarity scores were mainly an artefact of shared model error, both conditions converging on the same wrong or hedged answer, there is little reason to expect that pattern to hold this consistently across two architecturally distinct model families and three typologically distinct languages; different models tend to fail in different, model-specific ways, which would show up as more variable similarity scores across the Model factor than the near-zero effect actually observed ($\eta_p^2 = 0.001$, $p = .078$). A structural explanation fits the evidence better: Condition A's FOCUS axes (Section 3.2.3) were designed specifically to bias which topics a response emphasises, pharmacology, prognosis, cost, and so on, relative to Condition B's unweighted free-text prompt. A response built around a deliberately different topical emphasis is not an error; it is the FOCUS axes doing exactly what they were designed to do. Under this reading, a similarity score near 1.0 would have been the more concerning result, since it would suggest the FOCUS axes had no detectable effect on response content at all; a score of zero would suggest incoherent output. A moderate, stable score in between is closer to the expected signature of "same underlying answer, differently weighted emphasis" than of either failure mode in isolation.

This interpretation should be stated as the better-supported reading of the evidence available, not as something this thesis has proven. Response similarity, computed as embedding cosine similarity between two model outputs, has no access to an independent correctness signal; distinguishing "legitimately different emphasis" from "one or both responses are subtly wrong" with confidence would require expert clinical review of response content against the original query, which was outside this thesis's scope and is named directly as a limitation in Chapter 6. What this thesis can support is the narrower, more careful claim that a moderate, cross-model, cross-language-stable similarity score is consistent with the FOCUS-axis design mechanism functioning as intended, and is not, on the evidence available, more consistent with the alternative explanation of shared model failure.

## 5.4. Latency as an independent cost dimension

Section 4.4 established that latency does not track token cost. That result deserves discussion in its own right, because token cost and latency are not two measurements of the same underlying quantity; they are governed by different mechanisms, and a caching architecture built to reduce one has no obligation to reduce the other. Caching, as introduced in Section 2.3, avoids recomputing attention

states for a repeated prefix; it directly reduces compute and, on most providers' pricing models, directly reduces cost. It does nothing, by construction, to guarantee faster network round trips, shorter provider-side queueing, or faster generation of the output tokens that follow, all of which contribute to the wall-clock latency a user actually experiences, and none of which are downstream of the token-counting mechanism this thesis has otherwise treated as the central unit of cost. Model, not Condition, accounts for the overwhelming share of latency variance ($\eta_p^2 = 0.808$ versus $0.125$ for the Condition$\times$Model interaction; Table 4.4), and the two illustrative cases in Section 4.4 make the decoupling concrete: on Gemini 2.5 Flash-Lite for Hindi, Condition A is cheaper in tokens yet 25% slower; on GPT-5.4 for Hindi, Condition A is both more expensive and slower. It is moving in the same direction only because the same caching shortfall happens to touch both variables at once. It is deduced that this is not because the variables are linked in general.

This matters specifically for the domain this thesis targets. Section 2.4 established that consumer health queries are personal and often emotionally weighted; a user submitting a symptom-related question is plausibly more sensitive to a delayed response than a user in a lower-stakes interaction, since the wait itself can compound the anxiety the query was prompted by. If a structured interface were adopted specifically for its token-cost or energy-equity benefits (Section 2.2.1), and those benefits materialised exactly as intended, as they did on the Gemini variants tested, a designer could still end up shipping a slower experience for the user in the same interaction, since the two outcomes are not entailed by one another. Token efficiency is a real and defensible design goal in its own right, established across Sections 4.1 and 4.2; it is not, on this evidence, a proxy for perceived responsiveness, and should not be treated as one.

The latency measured throughout this chapter is model response latency only, the time between a fully formed query being submitted and a response returning. It is not the total interaction time a user experiences (In both conditions A & B), which additionally includes however long it takes to select an intent, position keywords, and adjust Kiviat vertices before submission. Whether that additional interaction time offsets, matches, or is trivial relative to the model-latency effects reported here is a genuinely open question this thesis did not measure and does not claim to answer; it is returned to as a limitation in Chapter 6.

## 5.5. Design implications

The preceding sections were written for a reader evaluating this thesis's claims. This section is written for a reader who might build something similar and asks what should actually change in how they design and evaluate it.

**Benchmark caching absorption before relying on it, per provider.** The single most consequential design assumption in this thesis, that a long, structured system prompt would be caching-eligible and therefore cheap to run repeatedly, held for one tokenizer family and failed for another, and nothing about the interface design itself signalled which outcome to expect in advance (Section 5.1). A designer building a similar cost-saving intervention should treat caching absorption as an empirical property to measure for each target provider, using something like the `processed_tokens` metric applied throughout this thesis (input tokens minus cached tokens; Section 3.2.3), rather than as a constant that follows automatically from a provider offering a caching feature at all.

**Do not treat token savings and latency as the same design goal.** Section 5.4 showed these move independently, and in the domain this thesis targets, personal, often anxious health queries, perceived responsiveness plausibly matters to a user in ways an abstract token count does not. A team optimising for one should measure the other directly rather than assuming it moves in the same direction, and should be explicit, including with end users, about which benefit, cost, energy, or speed, a given design change is actually intended to deliver.

# 6

# Limitations

This section develops the limitations flagged throughout this report at the point they first arose, rather than introducing new caveats here for the first time. They are grouped into three categories: scope decisions made in the benchmarking and dataset design, validity questions specific to the Medquery taxonomy and interface, and constraints on what this thesis's measurements can and cannot establish.

## 6.1. Data and benchmarking scope

The RQ1 benchmark and the Medquery taxonomy both derive from MeQSum, translated from English into Hindi and Arabic via the DeepL API (Section 3.1.5). Machine translation introduces its own token-length variance, differences in phrasing, segmentation, or register that a translation model introduces independently of anything the underlying tokenizer does, and this variance is confounded with the tokenizer effect RQ1 was designed to isolate. This thesis did not attempt to separate the two; the token tax reported in Section 4.1 should be read as the tax observed under this specific translation pipeline, not as a pipeline-independent quantity.

Model coverage is bounded to three commercial, general-purpose models across two tokenizer families (Section 3.1.3); locally optimised or language-specific tokenizers, which might narrow or eliminate the tax this thesis measures, were not tested and remain an open empirical question. Language coverage is bounded to English, Hindi, and Arabic, chosen deliberately for script and morphological diversity (Section 3.1.2), but three languages cannot establish that the token tax, or Medquery's mitigation of it, generalises to language families not represented here, particularly agglutinative or logographic scripts that differ from all three tested. Domain coverage is similarly bounded to consumer health queries; whether the same interface pattern transfers to other experiential, multi-dimensional query domains is a question this thesis motivates but does not test.

## 6.2. Design and taxonomy validity

The inductive coding procedure that produced the Medquery's taxonomy (Section 3.2.3) was LLM-assisted, with the researcher reviewing rather than independently re-deriving every classification. Any systematic bias in how the assisting model interpreted ambiguous clauses is a limitation carried into the taxonomy itself, and was not independently audited by a second human coder. Relatedly, the coded corpus contained zero instances of the Clinical Triage intent category (Section 3.2.3), plausibly a genuine feature of Medquery's origin as a non-urgent consumer information service rather than a flaw in the taxonomy, but it means this thesis's coverage claims have not been tested against the query type they would matter most for, and should not be assumed to extend to it.

The Kiviat interface itself carries a documented usability trade-off rather than an unambiguous advantage: Section 3.2.4 showed that radar-chart formats are, on the balance of the patient-facing visualisation literature, understood less reliably by patients than simpler formats such as bar charts or number lines. This thesis argued that trade-off was justified by the FELT/FOCUS structure's genuinely multivariate nature, but that argument was not tested with real users in this thesis; no usability study,

comprehension check, or learnability measure was collected for the Kiviat interface itself, and the claim remains a design rationale rather than an empirical finding.

## 6.3. Measurement validity

Three measurement gaps constrain how far this thesis's results can be pushed. First, response similarity (Section 4.3) is a measure of agreement between two model outputs, not of correctness against any independent reference; Section 5.3 argued that the observed pattern is more consistent with FOCUS-axis-driven emphasis differences than with shared model error, but this remains an inference from indirect evidence rather than a claim supported by expert clinical review, which this thesis did not conduct.

Second, latency was measured as model response time only (Section 4.4); the total interaction time a MedQuery user experiences, including the time spent forming a query through the interface itself, was not measured, and this thesis makes no claim about how the two combine. Relatedly, this thesis's environmental argument (Section 5.5) is inferred from the established relationship between token count and inference-time compute (Jegham et al., 2025; Strubell et al., 2019), not from direct measurement of energy draw or carbon emissions for the specific calls made in this thesis's own benchmarking; the direction of the claim is well-supported by prior work, but the magnitude of any actual emissions reduction from this thesis's own findings has not been independently quantified.

Third, the anomalously low coefficient of variation observed for GPT-5.4's Condition A calls on Arabic (0.025, against 0.43–0.57 for English and Hindi; Section 4.2) was interpreted as a signature of a stable but partial caching cutoff, but this interpretation was not verified against any caching-internals data, since none is exposed by the provider's API, it remains the best-supported reading of the available evidence rather than a confirmed mechanism.

None of these limitations undermine the central, large-effect finding of this thesis; the Condition$\times$Model interaction reported in Section 4.2 is far larger than any of the confounds discussed here and plausibly accounts for on its own. They do bound how far its secondary findings, and the taxonomy and interface built to test it, should be generalised beyond the specific models, languages, and domain this thesis examined.

# 7

# Conclusion

This thesis set out from a single observation: the token, the unit an LLM is trained on, priced by, and constrained by, is not counted the same way in every language, and the resulting penalty, the token tax, has so far been addressed only at the tokenizer and prompt-compression layers, never at the interface layer closest to the user (Section 2.7). Two research questions followed from that gap.

**RQ1** asked whether the token tax exists, at what magnitude, for consumer health queries across English, Hindi, and Arabic. It does: language has a significant effect on input token count ($F(2, 8803) = 95.85$, $p < .001$), with English consistently cheapest and Hindi and Arabic statistically indistinguishable from each other, but each significantly costlier than English, carrying the burden consistently across three commercial models spanning two tokenizer architectures (Section 4.1).

**RQ2** asked whether a visual-first, structured-query interface reduces that cost. The answer this thesis arrived at is more precise than the question as originally posed: it does, but only contingent on the target provider's caching architecture absorbing the interface's structured system prompt close to completely. That condition held for both Gemini variants tested, where MedQuery reduced processed-token cost by roughly 70 tokens per query; it did not hold for GPT-5.4, where the same design increased cost by roughly 309 tokens per query (Section 4.2). This is the central finding of this thesis, and it explains the direction of the effect in every condition tested.

This reframing is also this thesis's clearest demonstration of what a systems-thinking lens is for. The interface layer is genuine leverage against the token tax; RQ1 and RQ2 together establish that a problem originating at the tokenizer layer can be partially absorbed several layers downstream, at the one layer an independent designer can actually act on. But Section 5.2 showed that leverage is not self-contained: it is exercised through a caching layer the interface designer does not control and cannot inspect from outside, and its effect can only be predicted, or trusted, by tracing the full stack rather than the interface in isolation. A designer who took only the Gemini result at face value would have shipped a regression on GPT-5.4 without ever knowing why. This thesis's contribution is not only the interface itself, but the demonstration, empirical, not merely argued, that evaluating an interface-layer intervention against a multi-provider infrastructure requires exactly this kind of cross-layer verification as a matter of course.

The token tax, throughout this thesis, has been framed primarily as a financial and equity burden — and it is one. It is also an environmental one, for the same underlying reason: inference-time compute and energy scale with token count (Section 2.2), so every avoidable token is also avoidable emissions (Lundin et al., 2026; Solatorio et al., 2024), at a scale that is trivial per query and material in aggregate once multiplied across the query volumes real commercial services report (Section 5.5) (Jegham et al., 2025). Read this way, MedQuery's token saving on the Gemini models is not only a cost-reduction result; it is a proof of concept that interface design — normally understood as a usability or accessibility concern — can be a legitimate, measurable lever on the environmental footprint of an AI service, at exactly the layer this thesis argued was previously unexamined (Bieniek et al., 2024; Jegham et al., 2025). That this same lever reversed into a cost *and* emissions increase on GPT-5.4 is not a contradiction of

this claim but a restatement of it: the environmental benefit, like the financial one, is real only where the infrastructure beneath the interface is configured to realise it, which is precisely the systems-level contingency this thesis set out to characterise (Jegham et al., 2025).

## 7.1. Future scope

Six directions follow directly from the limitations named in Chapter 6. First, replicating RQ2 against a wider set of providers with differently implemented caching architectures, ideally including providers whose caching behaviour is documented rather than inferred, as GPT-5.4's had to be here, would establish whether the Gemini/GPT-5.4 split reflects two ends of a spectrum or a genuine binary. Second, extending both studies to languages outside the Latin/Devanagari/Perso-Arabic scripts tested here, particularly agglutinative and logographic languages, would test whether the token tax and MedQuery's mitigation of it hold at the extremes of morphological and script diversity rather than only across the three languages examined. Third, a controlled usability study of the Kiviat interface itself- comprehension, learnability, and completion time against the free-text baseline, with real users rather than the automated benchmarking used throughout this thesis- would close the gap between the design rationale offered in Section 3.2.4 and empirical evidence for or against it, and would additionally supply the total-interaction-time measure. Fourth, expert clinical review of response content across conditions would test the FOCUS-axis interpretation offered in Section 5.3 directly, rather than through the indirect evidence available from response similarity alone. Fifth, the parametric, non-verbal encoding approach developed here for health queries is not inherently health-specific; testing it against other experiential, multi-dimensional query domains, mental health check-ins, financial or legal intake, incident reporting, would indicate how far the underlying design pattern travels beyond the domain this thesis used to demonstrate it. Sixth, this thesis's environmental argument (Section 5.5) was inferred from token counts via an established but external compute-scaling relationship, not measured directly; instrumenting the benchmarking pipeline used throughout this thesis with actual energy draw or provider-reported carbon figures, where available, would convert that inference into a direct, thesis-specific quantification of the emissions this interface pattern actually avoids or adds.

Taken together, this thesis found a real, measurable, and previously unexamined leverage point against the token tax, and found, in the same analysis, exactly how contingent that leverage is on infrastructure decisions the interface layer cannot see. Both findings are the point.

# 8

# Use of Artificial Intelligence Tools

In the interest of transparency, this chapter documents the artificial intelligence tools used during the production of this thesis, the purpose each served, and where in the document their use is relevant.

## 8.1. Statement of Responsibility

Generative and assistive AI tools were used at various stages of this project as productivity aids, under the direct instruction, review, and verification of the author. **The research questions, study design, methodological decisions, analysis, interpretation, and conclusions presented in this thesis are the sole responsibility of the author.** No AI tool was used to generate research findings, make methodological decisions, or draw conclusions independently of the author's direction and judgment. Where AI tools contributed to code, translated text, or drafted prose, all such output was reviewed, tested, verified, and edited by the author before inclusion.

A distinction is drawn between two categories of AI use in this project:

1. **AI as a research tool**, assisting the author's own work (documented in Table 8.1 below).
2. **AI as the object of study** : The large language models (Gemini 2.5/3.1 Flash-Lite, GPT-5.4) benchmarked and evaluated as part of the experimental design in Chapters 3 and 4. These are not authorship tools; their use is fully documented as part of the Methodology and is not repeated here.

## 8.2. Tools Used

**Table 8.1:** AI tools used during the production of this thesis

| Tool | Provider | Purpose | Where used | Verification |
|---|---|---|---|---|
| Claude (Anthropic, 2026a) | Anthropic | Brainstorming methodological decisions; discussing statistical approach; drafting and debugging Python scripts for data benchmarking and analysis; paraphrasing and copyediting assistance for readability; literature grounding suggestions | Methodology (Ch. 3); code in Appendices A–C; general copyediting throughout | All code tested against real data before use; all methodological suggestions independently evaluated and adopted or rejected by the author; all prose edited and approved by the author |
| DeepL Translator (DeepL SE, 2026) | DeepL SE | Machine translation of the MeQSum consumer health question dataset from English into Hindi and Arabic for the RQ1 benchmark | Methodology, Section 3.1.5 (dataset preparation) | Translation used directly as experimental stimuli; translation-induced token-length variance acknowledged as a study limitation (Section 3.1.5) |
| NotebookLM (Google, 2026) | Google | Locating and summarizing candidate literature during the literature review process | Literature Review (Ch. 2) | All cited claims independently verified against original source publications before inclusion; no claim was cited on the basis of an AI-generated summary alone |

## 8.3. Tools Not Covered by This Statement

JASP (statistical analysis) and Figma (interface prototyping) were used extensively in this project but are not AI tools and are documented in the Methodology (Sections 3.1.7 and 3.2.4) instead of here.

# References


Ahia, O., Kumar, S., Gonen, H., Kasai, J., Mortensen, D., Smith, N., & Tsvetkov, Y. (2023, December). Do all languages cost the same? tokenization in the era of commercial language models. In H. Bouamor, J. Pino, & K. Bali (Eds.), *Proceedings of the 2023 conference on empirical methods in natural language processing* (pp. 9904–9923). Association for Computational Linguistics. https://doi.org/10.18653/v1/2023.emnlp-main.614

Anthropic. (2026a). Claude (sonnet 4.5) [large language model] [Accessed during thesis preparation]. https://www.anthropic.com/claude

Anthropic. (2026b). *Claude sonnet 5*. https://www.anthropic.com/claude/sonnet

Bales, R. F. (1950). *Interaction process analysis: A method for the study of small groups*. Addison-Wesley Press.

Ben Abacha, A., & Demner-Fushman, D. (2019). A question-entailment approach to question answering. *BMC Bioinformatics*, *20*(1). https://doi.org/10.1186/s12859-019-3119-4

Bieniek, J., Rahouti, M., & Verma, D. C. (2024). *Generative ai in multimodal user interfaces: Trends, challenges, and cross-platform adaptability*. arXiv: 2411.10234 [cs.HC]. https://arxiv.org/abs/2411.10234

Brooke, J. (1996). Sus: A 'quick and dirty' usability scale. In P. W. Jordan, B. Thomas, B. A. Weerdmeester, & I. L. McClelland (Eds.), *Usability evaluation in industry* (pp. 189–194). CRC Press.

Butler, A., et al. (2025). *Hybrid user interfaces: Past, present, and future of complementary cross-device interaction in mixed reality*. arXiv: 2509.05491 [cs.HC]. https://arxiv.org/abs/2509.05491

DeepL SE. (2026). Deepl translator [machine translation software] [Accessed during thesis preparation]. https://www.deepl.com

Gim, I., Chen, G., Lee, S.-s., Sarda, N., Khandelwal, A., & Zhong, L. (2024). *Prompt cache: Modular attention reuse for low-latency inference*. arXiv: 2311.04934 [cs.CL]. https://arxiv.org/abs/2311.04934

Google. (2026). Notebooklm [ai-powered research and note-taking tool] [Accessed during thesis preparation]. https://notebooklm.google

Goyal, N., Gao, C., Chaudhary, V., Chen, P.-J., Wenzek, G., Ju, D., Krishnan, S., Ranzato, M., Guzman, F., & Fan, A. (2021). *The flores-101 evaluation benchmark for low-resource and multilingual machine translation*. arXiv: 2106.03193 [cs.CL]. https://arxiv.org/abs/2106.03193

Jegham, N., Abdelatti, M., Koh, C. Y., Elmoubarki, L., & Hendawi, A. (2025). *How hungry is ai? benchmarking energy, water, and carbon footprint of llm inference*. arXiv: 2505.09598 [cs.CY]. https://arxiv.org/abs/2505.09598

Jiang, H., Wu, Q., Lin, C.-Y., Yang, Y., & Qiu, L. (2023). *Llmlingua: Compressing prompts for accelerated inference of large language models*. arXiv: 2310.05736 [cs.CL]. https://arxiv.org/abs/2310.05736

Jørgensen, A. H., & Myers, B. (2008). User interface history. *CHI '08 Extended Abstracts on Human Factors in Computing Systems*, 2415–2418. https://doi.org/10.1145/1358628.1358696

Kilicoglu, H., Ben Abacha, A., Mrabet, Y., Shooshan, S. E., Rodriguez, L., Masterton, K., & Demner-Fushman, D. (2018). Semantic annotation of consumer health questions. *BMC Bioinformatics*, *19*(1), 34. https://doi.org/10.1186/s12859-018-2045-1

Kim, T.-S., John Ignacio, M., Yu, S., Jin, H., & Kim, Y.-G. (2024). Ui/ux for generative ai: Taxonomy, trend, and challenge. *IEEE Access*, *12*, 179891–179911. https://doi.org/10.1109/ACCESS.2024.3502628

Korinek, A., & Stiglitz, J. E. (2026, March). *Steering technological progress* (Working Paper No. 34994). National Bureau of Economic Research. https://doi.org/10.3386/w34994

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J. E., Zhang, H., & Stoica, I. (2023). *Efficient memory management for large language model serving with pagedattention*. arXiv: 2309.06180 [cs.LG]. https://arxiv.org/abs/2309.06180

Lundin, J. M., Zhang, A., Karim, N., Louzan, H., Wei, G., Adelani, D. I., & Carroll, C. (2026). The token tax: Systematic bias in multilingual tokenization. *Proceedings of the 7th Workshop on African Natural Language Processing (AfricaNLP 2026)*, 103–112. https://doi.org/10.18653/v1/2026.africanlp-main.10

Meadows, D. H. (2008). *Thinking in systems: A primer* (D. Wright, Ed.). Chelsea Green Publishing.

Misra, A., Zamir, S. W., Hamidouche, W., Becker-Reshef, I., & Ferres, J. L. (2025). *Ai diffusion in low resource language countries*. arXiv: 2511.02752 [cs.CL]. https://arxiv.org/abs/2511.02752

Paruchuri, A., Aziz, M., Vartak, R., Ali, A., Uchehara, B., Liu, X., Chatterjee, I., & Agrawal, M. (2025, November). "what's up, doc?": Analyzing how users seek health information in large-scale conversational ai datasets. In C. Christodoulopoulos, T. Chakraborty, C. Rose, & V. Peng (Eds.), *Findings of the association for computational linguistics: Emnlp 2025* (pp. 2312–2336). Association for Computational Linguistics. https://doi.org/10.18653/v1/2025.findings-emnlp.125

Peng, X., Koch, J., & Mackay, W. E. (2024). Designprompt: Using multimodal interaction for design exploration with generative ai. *Proceedings of the 2024 ACM Designing Interactive Systems Conference*, 1–15. https://doi.org/10.1145/3643834.3661588

Petrov, A., La Malfa, E., Torr, P. H. S., & Bibi, A. (2023). *Language model tokenizers introduce unfairness between languages*. arXiv: 2305.15425 [cs.CL]. https://arxiv.org/abs/2305.15425

Pontiki, M., Galanis, D., Pavlopoulos, J., Papageorgiou, H., Androutsopoulos, I., & Manandhar, S. (2014, August). Semeval-2014 task 4: Aspect based sentiment analysis. In P. Nakov & T. Zesch (Eds.), *Proceedings of the 8th international workshop on semantic evaluation (semeval 2014)* (pp. 27–35). Association for Computational Linguistics. https://doi.org/10.3115/v1/S14-2004

Pool, J., Indulska, M., & Sadiq, S. (2024). Large language models and generative ai in telehealth: A responsible use lens. *Journal of the American Medical Informatics Association*, *31*(9), 2125–2136. https://doi.org/10.1093/jamia/ocae035

Sambasivan, N., Arnesen, E., Hutchinson, B., Doshi, T., & Prabhakaran, V. (2021). Re-imagining algorithmic fairness in india and beyond. *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*, 315–328. https://doi.org/10.1145/3442188.3445896

Sennrich, R., Haddow, B., & Birch, A. (2016, August). Neural machine translation of rare words with subword units. In K. Erk & N. A. Smith (Eds.), *Proceedings of the 54th annual meeting of the association for computational linguistics (volume 1: Long papers)* (pp. 1715–1725). Association for Computational Linguistics. https://doi.org/10.18653/v1/P16-1162

Shneiderman, B. (1983). Direct manipulation: A step beyond programming languages. *Computer*, *16*(8), 57–69. https://doi.org/10.1109/MC.1983.1654471

Solatorio, A. V., Vicente, G. S., Krambeck, H., & Dupriez, O. (2024). *Double jeopardy and climate impact in the use of large language models: Socio-economic disparities and reduced utility for non-english speakers*. arXiv: 2410.10665 [cs.CL]. https://arxiv.org/abs/2410.10665

Stevens, S. S. (1946). On the theory of scales of measurement. *Science*, *103*(2684), 677–680. https://doi.org/10.1126/science.103.2684.677

Strubell, E., Ganesh, A., & McCallum, A. (2019). *Energy and policy considerations for deep learning in nlp*. arXiv: 1906.02243 [cs.CL]. https://arxiv.org/abs/1906.02243

Sweller, J. (1988). Cognitive load during problem solving: Effects on learning. *Cognitive Science*, *12*(2), 257–285.

Thaker, N. G., Ali, T. N., Porter, M. E., Feeley, T. W., Kaplan, R. S., & Frank, S. J. (2016). Communicating value in health care using radar charts: A case study of prostate cancer. *Journal of Oncology Practice*, *12*(9), 813–820. https://doi.org/10.1200/JOP.2016.011320

Toraman, C., Yilmaz, E. H., Şahiuç, F., & Ozcelik, O. (2023). Impact of tokenization on language models: An analysis for turkish. *ACM Transactions on Asian and Low-Resource Language Information Processing*, *22*(4), 1–21. https://doi.org/10.1145/3578707

Turchioe, M., Myers, A., Isaac, S., Baik, D., Grossman, L., Ancker, J., & Masterson Creber, R. (2019). A systematic review of patient-facing visualizations of personal health data. *Applied Clinical Informatics*, *10*(4), 751–770. https://doi.org/10.1055/s-0039-1697592

Yang, Z., Xu, X., Yao, B., Zhang, S., Rogers, E., Intille, S., Shara, N., Gao, G. G., & Wang, D. (2024). *Talk2care: Facilitating asynchronous patient-provider communication with large-language-model*. arXiv: 2309.09357 [cs.CL]. https://arxiv.org/abs/2309.09357

Yun, H. S., Bickmore, T., et al. (2025). Online health information-seeking in the era of large language models: Cross-sectional web-based survey study. *Journal of Medical Internet Research*, *27*, e68560.

# A

# Script used for Translation

```
1
2
3
4  import os
5  import time
6  import logging
7  import random
8  import pandas as pd
9  from tqdm import tqdm
10 from dotenv import load_dotenv
11
12 try:
13     import deepl
14 except ImportError:
15     raise ImportError("Install DeepL: pip install deepl")
16
17 #   Configuration
18
19 load_dotenv()
20
21 DEEPL_API_KEY = os.getenv("DEEPL_API_KEY")
22 if not DEEPL_API_KEY:
23     raise EnvironmentError(
24         "DEEPL_API_KEY not found. Add it to your .env file.\n"
25         "Get a free API key at: https://www.deepl.com/pro-api"
26     )
27
28
29 TARGET_LANGUAGES = {
30     "hi": "HI",   # Hindi (Devanagari script)
31     "ar": "AR",   # Arabic (Modern Standard Arabic)
32 }
33
34 # Stratification thresholds (in English token count, approx.)
35 # Short: < 15 tokens (~60 chars), Medium: 15-40, Long: > 40
36 COMPLEXITY_THRESHOLDS = {
37     "short":  (0, 60),      # character-based approximation
38     "medium": (60, 160),
39     "long":   (160, 9999),
40 }
41
42 # Back-translation sample: 10% stratified
43 BACK_TRANSLATION_SAMPLE_RATE = 0.10
```

```

DRIFT_THRESHOLD = 0.35

# Output paths
DATA_DIR = "./data"
OUTPUT_PATH       = f"{DATA_DIR}/meqsum_translated.csv"
BACK_TRANS_PATH   = f"{DATA_DIR}/back_translation_sample.csv"
REPORT_PATH       = f"{DATA_DIR}/translation_report.txt"
LOG_PATH          = f"{DATA_DIR}/translation_run.log"

# Rate limiting - DeepL free tier: 500k chars/month
# Paid tier is generous; set conservative delay for stability
DELAY_BETWEEN_CALLS = 0.2  # seconds

#   Logging

os.makedirs(DATA_DIR, exist_ok=True)

logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s [%(levelname)s] %(message)s",
    handlers=[
        logging.FileHandler(LOG_PATH),
        logging.StreamHandler()
    ]
)
logger = logging.getLogger(__name__)

#   DeepL Client

translator = deepl.Translator(DEEPL_API_KEY)

#   Helper Functions

def classify_complexity(text: str) -> str:

    n = len(text.strip())
    if n <= COMPLEXITY_THRESHOLDS["short"][1]:
        return "short"
    elif n <= COMPLEXITY_THRESHOLDS["medium"][1]:
        return "medium"
    else:
        return "long"


def translate_text(text: str, target_lang: str,
                   max_retries: int = 3) -> str | None:

    for attempt in range(1, max_retries + 1):
        try:
            result = translator.translate_text(
                text,
                source_lang="EN",
                target_lang=target_lang,
            )
            return result.text
        except Exception as e:
            logger.warning(
                f"DeepL error (attempt {attempt}/{max_retries}): {e}"
            )
```

```
            if attempt < max_retries:
                time.sleep(attempt * 2)
    return None


def back_translate(text: str, intermediate_lang: str) -> str | None:

    # Step 1: EN → target language
    intermediate = translate_text(text, intermediate_lang)
    if not intermediate:
        return None

    time.sleep(DELAY_BETWEEN_CALLS)

    # Step 2: target language → EN
    back = translate_text(intermediate, "EN-US")
    return back


def compute_drift_score(original: str, back_translated: str) -> float:

    if not back_translated:
        return 1.0
    orig = set(original.lower().split())
    back = set(back_translated.lower().split())
    if not orig:
        return 1.0
    overlap = len(orig & back) / len(orig)
    return round(1.0 - overlap, 3)


def select_stratified_sample(df: pd.DataFrame,
                             rate: float = 0.10) -> pd.DataFrame:

    sample_frames = []
    for tier in ["short", "medium", "long"]:
        tier_df = df[df["complexity_tier"] == tier]
        n = max(1, int(len(tier_df) * rate))
        sample_frames.append(
            tier_df.sample(n=min(n, len(tier_df)), random_state=42)
        )
    return pd.concat(sample_frames).reset_index(drop=True)


#   Main Pipeline

def run_translation_pipeline(meqsum_path: str,
                             pilot_mode: bool = True,
                             pilot_n: int = 50):


    logger.info("=" * 60)
    logger.info("TRANSLATION PIPELINE - TU Delft MSc Thesis")
    logger.info("=" * 60)

    #   Load MeQSum
    logger.info(f"Loading MeQSum from: {meqsum_path}")
    df = pd.read_csv(meqsum_path)

    # Auto-detect query column
    # MeQSum typically uses 'Question' or 'NLM_QUESTION'
```

```
    query_col = None
    for candidate in ["NLM_QUESTION", "Question", "question",
                      "query", "Query"]:
        if candidate in df.columns:
            query_col = candidate
            break

    if not query_col:
        raise ValueError(
            f"Cannot find query column. Available: {df.columns.tolist()}\n"
            "Update the `candidate` list in auto-detect block."
        )

    logger.info(f"Using query column: '{query_col}'")
    logger.info(f"Total queries in dataset: {len(df)}")

    # Standardise columns
    df = df.rename(columns={query_col: "query_en"})
    if "query_id" not in df.columns:
        df["query_id"] = [f"MEQ_{i:04d}" for i in range(len(df))]

    # Classify complexity
    df["complexity_tier"] = df["query_en"].apply(classify_complexity)
    tier_counts = df["complexity_tier"].value_counts()
    logger.info(f"Complexity distribution:\n{tier_counts.to_string()}")

    # Pilot mode
    if pilot_mode:
        df = df.sample(n=pilot_n, random_state=42)
        logger.info(f"PILOT MODE: Running on {pilot_n} queries.")
    else:
        logger.info("FULL MODE: Translating all queries.")

    #   Resume Support
    if os.path.exists(OUTPUT_PATH):
        df_existing = pd.read_csv(OUTPUT_PATH)
        completed_ids = set(df_existing["query_id"])
        df = df[~df["query_id"].isin(completed_ids)]
        logger.info(
            f"Resuming: {len(completed_ids)} already done, "
            f"{len(df)} remaining."
        )
        df_results = df_existing
    else:
        df_results = pd.DataFrame()

    #   Translation Loop
    new_rows = []
    total = len(df) * len(TARGET_LANGUAGES)

    with tqdm(total=total, desc="Translating") as pbar:
        for _, row in df.iterrows():
            query_id  = row["query_id"]
            query_en  = row["query_en"]
            tier      = row["complexity_tier"]

            translated = {
                "query_id":        query_id,
                "query_en":        query_en,
                "complexity_tier": tier,
            }
```

```
227
228             for lang_code, deepl_code in TARGET_LANGUAGES.items():
229                 col = f"query_{lang_code}"
230                 translation = translate_text(query_en, deepl_code)
231                 translated[col] = translation
232
233                 if not translation:
234                     logger.warning(
235                         f"Translation failed: {query_id} → {lang_code}"
236                     )
237
238                 time.sleep(DELAY_BETWEEN_CALLS)
239                 pbar.update(1)
240
241             new_rows.append(translated)
242
243             # Save incrementally
244             df_results = pd.concat(
245                 [df_results, pd.DataFrame([translated])],
246                 ignore_index=True
247             )
248             df_results.to_csv(OUTPUT_PATH, index=False)
249
250     logger.info(f"Translation complete. Saved to: {OUTPUT_PATH}")
251     logger.info(f"Total rows: {len(df_results)}")
252
253     #   Back-Translation Validation
254     logger.info("Running back-translation validation on 10% sample...")
255
256     sample_df = select_stratified_sample(df_results,
257                                          BACK_TRANSLATION_SAMPLE_RATE)
258     logger.info(f"Back-translation sample size: {len(sample_df)}")
259
260     bt_rows = []
261     for _, row in tqdm(sample_df.iterrows(),
262                        total=len(sample_df),
263                        desc="Back-translating"):
264         bt_row = {
265             "query_id":        row["query_id"],
266             "query_en":        row["query_en"],
267             "complexity_tier": row["complexity_tier"],
268         }
269
270         for lang_code, deepl_code in TARGET_LANGUAGES.items():
271             translated = row.get(f"query_{lang_code}")
272             if not translated:
273                 continue
274
275             # Back-translate to English
276             bt_text = back_translate(row["query_en"], deepl_code)
277             drift   = compute_drift_score(row["query_en"], bt_text)
278
279             bt_row[f"back_translated_{lang_code}"] = bt_text
280             bt_row[f"drift_score_{lang_code}"]     = drift
281             bt_row[f"flagged_{lang_code}"]         = drift > DRIFT_THRESHOLD
282
283             time.sleep(DELAY_BETWEEN_CALLS)
284
285         bt_rows.append(bt_row)
286
287     df_bt = pd.DataFrame(bt_rows)
```

```
    df_bt.to_csv(BACK_TRANS_PATH, index=False)

    #   Quality Report
    report_lines = [
        "TRANSLATION QUALITY REPORT",
        "=" * 50,
        f"Dataset: MeQSum",
        f"Total translated: {len(df_results)}",
        f"Pilot mode: {pilot_mode}",
        f"Back-translation sample: {len(df_bt)}",
        "",
    ]

    for lang_code in TARGET_LANGUAGES:
        col_flag  = f"flagged_{lang_code}"
        col_drift = f"drift_score_{lang_code}"

        if col_drift in df_bt.columns:
            mean_drift = df_bt[col_drift].mean()
            n_flagged  = df_bt[col_flag].sum() if col_flag in df_bt.columns else 0
            failed     = df_results[f"query_{lang_code}"].isna().sum()

            report_lines += [
                f"Language: {lang_code.upper()}",
                f"  Mean drift score:    {mean_drift:.3f}",
                f"  Flagged for review:  {n_flagged} "
                f"({100*n_flagged/len(df_bt):.1f}%)",
                f"  Translation failures:{failed}",
                f"  Action required: Human review of "
                f"{n_flagged} flagged queries",
                "",
            ]

    report_lines += [
        "NEXT STEPS:",
        "1. Review flagged queries in back_translation_sample.csv",
        "2. Send flagged Hindi queries to native speaker validator",
        "3. For Arabic, verify MSA naturalness on a sample",
        "4. Document final validation statistics for thesis Section 3.2",
        "5. Run token_tax_benchmark.py with meqsum_translated.csv",
    ]

    report_text = "\n".join(report_lines)
    with open(REPORT_PATH, "w") as f:
        f.write(report_text)

    logger.info("\n" + report_text)
    logger.info(f"\nBack-translation data: {BACK_TRANS_PATH}")
    logger.info(f"Quality report: {REPORT_PATH}")

    return df_results, df_bt


#   Entry Point

if __name__ == "__main__":
    # STEP 1: Download MeQSum dataset from:
    #   https://github.com/abachaa/MeQSum
    # File: MeQSum_ACL2019_BenAbacha_Demner-Fushman.xlsx
    # Convert to CSV before running this script.

```

```
    MEQSUM_PATH = "./data/meqsum_raw.csv"



    PILOT_MODE = True  # ← Change to False for full run

    df_translated, df_validation = run_translation_pipeline(
        meqsum_path=MEQSUM_PATH,
        pilot_mode=PILOT_MODE,
        pilot_n=50,
    )
```

# B

# Script used for Research Question 1

Full source of the token-tax benchmarking script used to collect the RQ1 dataset (Chapter 3.). See Section 3.1 for the experimental procedure this script implements.

```
1
2
3
4  import os
5  import time
6  import logging
7  import pandas as pd
8  from tqdm import tqdm
9  from dotenv import load_dotenv
10
11 #   API Clients
12
13 load_dotenv()
14
15 GEMINI_API_KEY = os.getenv("GEMINI_API_KEY")
16 OPENAI_API_KEY = os.getenv("OPENAI_API_KEY")
17
18 if not GEMINI_API_KEY:
19     raise EnvironmentError("GEMINI_API_KEY not found in .env file.")
20
21 # Gemini client - always needed
22 from google import genai
23 from google.genai import types
24 gemini_client = genai.Client(api_key=GEMINI_API_KEY)
25
26 # OpenAI client - only needed if gpt54 is in MODELS
27 openai_client = None
28 if OPENAI_API_KEY:
29     try:
30         from openai import OpenAI
31         openai_client = OpenAI(api_key=OPENAI_API_KEY)
32         print("   OpenAI client initialised.")
33     except ImportError:
34         print("   openai package not installed. GPT-5.4 will be skipped.")
35         print("    Install: pip install openai --break-system-packages")
36 else:
37     print("   OPENAI_API_KEY not found. GPT-5.4 will be skipped.")
38
39
40
```

```python
MODELS = {
    #   Gemini 2.5 (stable - already pilot-tested)
    "gemini_25_flash_lite":  "gemini-2.5-flash-lite",

    #   Gemini 3.1 (preview - new generation)
    "gemini_31_flash_lite":  "gemini-3.1-flash-lite-preview",
    "gemini_31_pro":         "gemini-3.1-pro-preview",

    #   OpenAI (cross-provider proof)
    "gpt54":                 "gpt-5.4",
}


MODELS_THINKING_SUPPORT = {
    "gemini_25_flash_lite":  False,
    "gemini_31_flash_lite":  True,
    "gemini_31_pro":         True,
    "gpt54":                 False,
}

# Which provider each model uses
MODEL_PROVIDER = {
    "gemini_25_flash_lite":  "gemini",
    "gemini_31_flash_lite":  "gemini",
    "gemini_31_pro":         "gemini",
    "gpt54":                 "openai",
}

#   Thinking Configuration

def is_gemini3(model_key: str) -> bool:

    return model_key.startswith("gemini_3")


def build_thinking_config(model_key: str, thinking_mode: str):

    # Models that require thinking - cannot use budget=0
    REQUIRES_THINKING = {"gemini_31_pro", "gemini_31_flash_lite"}

    if thinking_mode == "thinking_off" and model_key in REQUIRES_THINKING:
        budget = 1      # minimum valid budget - effectively thinking off
    elif thinking_mode == "thinking_off":
        budget = 0      # Gemini 2.5 - full thinking disable
    else:
        budget = 1024   # thinking on for all models

    try:
        return types.ThinkingConfig(thinking_budget=budget)
    except Exception:
        logger.warning(
            f"ThinkingConfig failed for {model_key}/{thinking_mode}. "
            f"Running without thinking config."
        )
        return None


#   Language Configuration

LANGUAGES = {
    "en": "query_en",
```

```
    "hi": "query_hi",
    "ar": "query_ar",
}



SYSTEM_PROMPT = (
    "You are a consumer health information assistant. "
    "Answer the user's health question clearly and concisely. "
    "Do not provide a diagnosis. Recommend consulting a doctor if needed."
)



RATE_LIMIT_DELAY_SECONDS = {
    "gemini_25_flash_lite":  3.5,
    "gemini_31_flash_lite":  4.0,
    "gemini_31_pro":         30.0,
    "gpt54":                 3.0,
}

#   Output Paths

OUTPUT_DIR       = "./results"
RAW_RESULTS_PATH = f"{OUTPUT_DIR}/raw_benchmark_results.csv"
SUMMARY_PATH     = f"{OUTPUT_DIR}/token_tax_summary.csv"
LOG_PATH         = f"{OUTPUT_DIR}/benchmark_run.log"



PRICING = {
    "gemini_25_flash_lite": {
        "input_per_1m":    0.10,
        "output_per_1m":   0.40,
        "thinking_per_1m": 0.00,
    },
    "gemini_31_flash_lite": {
        "input_per_1m":    0.25,
        "output_per_1m":   1.50,
        "thinking_per_1m": 0.25,
    },
    "gemini_31_pro": {
        "input_per_1m":    1.25,
        "output_per_1m":   10.00,
        "thinking_per_1m": 3.50,
    },
    "gpt54": {
        "input_per_1m":    2.50,
        "output_per_1m":   15.00,
        "thinking_per_1m": 0.00,
    },
}

#   Logging

os.makedirs(OUTPUT_DIR, exist_ok=True)

logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s [%(levelname)s] %(message)s",
    handlers=[
```

```
        logging.FileHandler(LOG_PATH),
        logging.StreamHandler()
    ]
)
logger = logging.getLogger(__name__)


#   Gemini Query Function

def query_gemini(
    model_key: str,
    query_text: str,
    query_id: str,
    language: str,
    thinking_mode: str = "thinking_off",
    max_retries: int = 3
) -> dict:

    model_name = MODELS[model_key]

    REQUIRES_THINKING = {"gemini_31_pro", "gemini_31_flash_lite"}
    # Relabel thinking_off as thinking_minimal for models that require thinking
    effective_thinking_mode = (
        "thinking_minimal"
        if thinking_mode == "thinking_off" and model_key in REQUIRES_THINKING
        else thinking_mode
    )

    result = {
        "query_id":        query_id,
        "language":        language,
        "model_key":       model_key,
        "model_name":      model_name,
        "provider":        "gemini",
        "thinking_mode":   effective_thinking_mode,
        "input_tokens":    None,
        "output_tokens":   None,
        "thinking_tokens": None,
        "total_tokens":    None,
        "latency_ms":      None,
        "response_text":   None,
        "error":           None,
    }

    for attempt in range(1, max_retries + 1):
        try:
            start_time     = time.perf_counter()
            thinking_config = build_thinking_config(model_key, thinking_mode)


            temp = 0.2 if thinking_mode == "thinking_off" else None

            config_kwargs = dict(
                system_instruction=SYSTEM_PROMPT,
                max_output_tokens=512,
            )
            # Only add thinking_config if it was successfully built
            if thinking_config is not None:
                config_kwargs["thinking_config"] = thinking_config
            if temp is not None:
                config_kwargs["temperature"] = temp
```

```
224
225             response = gemini_client.models.generate_content(
226                 model=model_name,
227                 contents=query_text,
228                 config=types.GenerateContentConfig(**config_kwargs),
229             )
230
231             latency_ms      = (time.perf_counter() - start_time) * 1000
232             usage           = response.usage_metadata
233             thinking_tokens = getattr(usage, "thoughts_token_count", 0) or 0
234
235             result["input_tokens"]    = usage.prompt_token_count
236             result["output_tokens"]   = usage.candidates_token_count
237             result["thinking_tokens"] = thinking_tokens
238             result["total_tokens"]    = usage.total_token_count
239             result["latency_ms"]      = round(latency_ms, 2)
240             result["response_text"]   = response.text[:300] if response.text else
                    ""
241             result["error"]           = None
242             break
243
244         except Exception as e:
245             logger.warning(
246                 f"Gemini attempt {attempt}/{max_retries} | "
247                 f"query={query_id} lang={language} "
248                 f"model={model_key} thinking={thinking_mode} | {e}"
249             )
250             if attempt == max_retries:
251                 result["error"] = str(e)
252             else:
253                 time.sleep(attempt * 4)
254
255     return result
256
257
258 #   OpenAI Query Function
259
260 def query_openai(
261     model_key: str,
262     query_text: str,
263     query_id: str,
264     language: str,
265     max_retries: int = 3
266 ) -> dict:
267
268     model_name = MODELS[model_key]
269
270     result = {
271         "query_id":        query_id,
272         "language":        language,
273         "model_key":       model_key,
274         "model_name":      model_name,
275         "provider":        "openai",
276         "thinking_mode":   "thinking_off",   # GPT-5.4 has no explicit thinking
                mode
277         "input_tokens":    None,
278         "output_tokens":   None,
279         "thinking_tokens": 0,                # not separately tracked for GPT-5.4
280         "total_tokens":    None,
281         "latency_ms":      None,
282         "response_text":   None,
```

```
        "error":           None,
    }

    if openai_client is None:
        result["error"] = "OpenAI client not initialised - check OPENAI_API_KEY"
        return result

    for attempt in range(1, max_retries + 1):
        try:
            start_time = time.perf_counter()

            response = openai_client.chat.completions.create(
                model=model_name,
                messages=[
                    {"role": "system", "content": SYSTEM_PROMPT},
                    {"role": "user",   "content": query_text},
                ],
                max_completion_tokens=512,  # GPT-5.x requires
                    max_completion_tokens
                temperature=0.2,
            )

            latency_ms = (time.perf_counter() - start_time) * 1000
            usage      = response.usage

            result["input_tokens"]  = usage.prompt_tokens
            result["output_tokens"] = usage.completion_tokens
            result["total_tokens"]  = usage.total_tokens
            result["latency_ms"]    = round(latency_ms, 2)
            result["response_text"] = (
                response.choices[0].message.content[:300]
                if response.choices else ""
            )
            result["error"] = None
            break

        except Exception as e:
            logger.warning(
                f"OpenAI attempt {attempt}/{max_retries} | "
                f"query={query_id} lang={language} "
                f"model={model_key} | {e}"
            )
            if attempt == max_retries:
                result["error"] = str(e)
            else:
                time.sleep(attempt * 4)

    return result


#   Dispatch Function

def query_model(
    model_key: str,
    query_text: str,
    query_id: str,
    language: str,
    thinking_mode: str = "thinking_off"
) -> dict:
    """Route query to the correct provider function."""
    provider = MODEL_PROVIDER.get(model_key, "gemini")
```

```
    if provider == "openai":
        return query_openai(model_key, query_text, query_id, language)
    else:
        return query_gemini(
            model_key, query_text, query_id, language, thinking_mode
        )


#   Token Ratio Calculator

def compute_token_ratios(df: pd.DataFrame) -> pd.DataFrame:

    en_baseline = (
        df[df["language"] == "en"]
        [["query_id", "model_key", "thinking_mode",
          "input_tokens", "total_tokens"]]
        .rename(columns={
            "input_tokens": "en_input_tokens",
            "total_tokens": "en_total_tokens",
        })
    )

    df = df.merge(
        en_baseline,
        on=["query_id", "model_key", "thinking_mode"],
        how="left"
    )

    df["token_ratio_input"] = df["input_tokens"] / df["en_input_tokens"]
    df["token_ratio_total"] = df["total_tokens"] / df["en_total_tokens"]
    df["thinking_dilution"] = (
        df["token_ratio_total"] / df["token_ratio_input"]
    )

    return df


#   Cost Estimator

def compute_cost_estimate(row: pd.Series) -> float:
    """Estimate USD cost including thinking token charges where applicable."""
    pricing = PRICING.get(row["model_key"], {})
    if not pricing:
        return 0.0

    input_cost    = ((row["input_tokens"]    or 0) / 1e6) * pricing["input_per_1m"
        ]
    output_cost   = ((row["output_tokens"]   or 0) / 1e6) * pricing["output_per_1m
        "]
    thinking_cost = ((row["thinking_tokens"] or 0) / 1e6) * pricing["
        thinking_per_1m"]

    return round(input_cost + output_cost + thinking_cost, 8)


#   Main Benchmark Orchestrator

def run_benchmark(dataset_path: str, sample_size: int = None):

    logger.info("=" * 65)
    logger.info("TOKEN TAX BENCHMARK v3 - TU Delft MSc Thesis")
```

```
    logger.info("Multi-provider: Gemini 2.5 + 3.1 + GPT-5.4")
    logger.info("=" * 65)

    # Load dataset
    try:
        df_queries = pd.read_csv(dataset_path, encoding='utf-8')
    except UnicodeDecodeError:
        df_queries = pd.read_csv(dataset_path, encoding='latin-1')

    required = ["query_id", "query_en", "query_hi", "query_ar"]
    missing  = [c for c in required if c not in df_queries.columns]
    if missing:
        raise ValueError(
            f"Dataset missing columns: {missing}\n"
            f"Available: {df_queries.columns.tolist()}\n"
            f"Run translate_meqsum.py first."
        )

    if sample_size:
        df_queries = df_queries.sample(n=sample_size, random_state=42)
        logger.info(f"Sample mode: {sample_size} queries.")
    else:
        logger.info(f"Full run: {len(df_queries)} queries.")

    #   Resume support
    all_results = []
    completed   = set()

    if os.path.exists(RAW_RESULTS_PATH):
        df_existing = pd.read_csv(RAW_RESULTS_PATH)
        all_results = df_existing.to_dict("records")
        completed   = set(zip(
            df_existing["query_id"].astype(str),
            df_existing["language"],
            df_existing["model_key"],
            df_existing["thinking_mode"],
        ))
        logger.info(f"Resuming: {len(completed)} results already recorded.")

    #   Build run plan
    run_plan = []
    for model_key in MODELS:
        # Skip OpenAI models if client not available
        if MODEL_PROVIDER[model_key] == "openai" and openai_client is None:
            logger.warning(f"Skipping {model_key} - OpenAI client not available.")
            continue

        supports_thinking = MODELS_THINKING_SUPPORT[model_key]
        modes = ["thinking_off", "thinking_on"] if supports_thinking else ["
            thinking_off"]

        for thinking_mode in modes:
            for lang_code in LANGUAGES:
                run_plan.append((model_key, thinking_mode, lang_code))

    total_calls = len(df_queries) * len(run_plan)
    logger.info(f"Run plan: {len(run_plan)} conditions")
    logger.info(f"Total API calls planned: {total_calls:,}")

    # Estimate runtime
    total_time_estimate = sum(
```

```
        len(df_queries) * len(LANGUAGES) *
        (2 if MODELS_THINKING_SUPPORT.get(mk, False) else 1) *
        RATE_LIMIT_DELAY_SECONDS.get(mk, 4.0)
        for mk in MODELS
        if not (MODEL_PROVIDER.get(mk) == "openai" and openai_client is None)
    )
    logger.info(
        f"Estimated runtime: ~{total_time_estimate/3600:.1f} hours "
        f"(recommend running overnight)"
    )

    #   Main loop
    with tqdm(total=total_calls, desc="Benchmarking") as pbar:
        for _, row in df_queries.iterrows():
            query_id = str(row["query_id"])

            for model_key, thinking_mode, lang_code in run_plan:
                pbar.update(1)

                # Skip if already done (resume logic)
                if (query_id, lang_code, model_key, thinking_mode) in completed:
                    continue

                col_name   = LANGUAGES[lang_code]
                query_text = str(row[col_name])

                # Skip rows where translation is missing
                if not query_text or query_text == "nan":
                    logger.warning(
                        f"Missing translation: {query_id} {lang_code} - skipping"
                    )
                    pbar.update(1)
                    continue

                result = query_model(
                    model_key=model_key,
                    query_text=query_text,
                    query_id=query_id,
                    language=lang_code,
                    thinking_mode=thinking_mode,
                )
                all_results.append(result)

                # Incremental save after every call - never lose data
                pd.DataFrame(all_results).to_csv(
                    RAW_RESULTS_PATH, index=False
                )

                delay = RATE_LIMIT_DELAY_SECONDS.get(model_key, 4.0)
                time.sleep(delay)

    logger.info("Benchmark loop complete. Computing derived metrics...")

    #   Post-processing
    df_results = pd.DataFrame(all_results)
    df_success = df_results[df_results["error"].isna()].copy()
    failed_rows = df_results["error"].notna().sum()

    if len(df_success) == 0:
        logger.error("No successful results to analyse.")
        return df_results, pd.DataFrame()
```

```
522
523     df_success = compute_token_ratios(df_success)
524     df_success["cost_usd"] = df_success.apply(compute_cost_estimate, axis=1)
525
526     enriched_path = RAW_RESULTS_PATH.replace(".csv", "_enriched.csv")
527     df_success.to_csv(enriched_path, index=False)
528
529     #   Summary table
530     summary = df_success.groupby(
531         ["model_key", "provider", "language", "thinking_mode"]
532     ).agg(
533         n_queries             = ("query_id",           "count"),
534         mean_input_tokens     = ("input_tokens",        "mean"),
535         mean_output_tokens    = ("output_tokens",       "mean"),
536         mean_thinking_tokens  = ("thinking_tokens",     "mean"),
537         mean_total_tokens     = ("total_tokens",        "mean"),
538         mean_ratio_input      = ("token_ratio_input",   "mean"),
539         mean_ratio_total      = ("token_ratio_total",   "mean"),
540         mean_thinking_dilution= ("thinking_dilution",   "mean"),
541         mean_latency_ms       = ("latency_ms",          "mean"),
542         p95_latency_ms        = ("latency_ms",          lambda x: x.quantile(0.95)
            ),
543         total_cost_usd        = ("cost_usd",            "sum"),
544     ).round(4).reset_index()
545
546     summary.to_csv(SUMMARY_PATH, index=False)
547     logger.info(f"Summary saved: {SUMMARY_PATH}")
548
549     #   Print key findings
550     logger.info("=" * 65)
551     logger.info("KEY FINDINGS - TOKEN TAX ACROSS PROVIDERS")
552     logger.info("=" * 65)
553
554     for model_key in df_success["model_key"].unique():
555         model_data = df_success[
556             (df_success["model_key"] == model_key) &
557             (df_success["language"] != "en") &
558             (df_success["thinking_mode"] == "thinking_off")
559         ]
560         if model_data.empty:
561             continue
562         logger.info(f"\n{model_key}:")
563         for lang in ["hi", "ar"]:
564             lang_data = model_data[model_data["language"] == lang]
565             if lang_data.empty:
566                 continue
567             mean_input = lang_data["token_ratio_input"].mean()
568             mean_total = lang_data["token_ratio_total"].mean()
569             mean_lat   = lang_data["latency_ms"].mean()
570             logger.info(
571                 f"  {lang.upper()}: input_ratio={mean_input:.3f}x  "
572                 f"total_ratio={mean_total:.3f}x  "
573                 f"latency={mean_lat:.0f}ms"
574             )
575
576     #   Cross-provider consistency check
577     logger.info("\ n CROSS-PROVIDER CONSISTENCY (systemic claim)     ")
578     logger.info("If ratios are consistent across providers, the Token Tax")
579     logger.info("is confirmed as a structural tokenizer property.")
580
581     for lang in ["hi", "ar"]:
```

```
        logger.info(f"\n  Language: {lang.upper()}")
        for model_key in df_success["model_key"].unique():
            data = df_success[
                (df_success["model_key"] == model_key) &
                (df_success["language"] == lang) &
                (df_success["thinking_mode"] == "thinking_off")
            ]
            if not data.empty:
                ratio = data["token_ratio_input"].mean()
                logger.info(f"    {model_key:30s}: {ratio:.3f}x")

    #   Thinking dilution analysis
    logger.info("\ n THINKING TOKEN CRITIQUE RESPONSE          ")
    thinking_models = [m for m in df_success["model_key"].unique()
                       if MODELS_THINKING_SUPPORT.get(m, False)]

    for model_key in thinking_models:
        off_data = df_success[
            (df_success["model_key"] == model_key) &
            (df_success["language"] != "en") &
            (df_success["thinking_mode"] == "thinking_off")
        ]
        on_data = df_success[
            (df_success["model_key"] == model_key) &
            (df_success["language"] != "en") &
            (df_success["thinking_mode"] == "thinking_on")
        ]
        if off_data.empty or on_data.empty:
            continue

        ratio_off = off_data["token_ratio_input"].mean()
        ratio_on  = on_data["token_ratio_total"].mean()
        dilution  = ratio_on / ratio_off if ratio_off > 0 else 0

        logger.info(f"\n  {model_key}:")
        logger.info(f"    Input ratio (thinking off): {ratio_off:.3f}x")
        logger.info(f"    Total ratio (thinking on):  {ratio_on:.3f}x")
        logger.info(f"    Dilution factor:            {dilution:.3f}")

        if dilution < 0.7:
            logger.info(" →    SUBSTANTIAL dilution. Foreground non-thinking
                results.")
        elif dilution < 0.9:
            logger.info(" →    MODERATE dilution. Report both ratios.")
        else:
            logger.info(" →    NEGLIGIBLE dilution. Full strength argument.")

    #   Study footprint
    total_tokens = df_success["total_tokens"].sum()
    total_cost   = df_success["cost_usd"].sum()

    logger.info("\n" + "=" * 65)
    logger.info(f"STUDY FOOTPRINT: {total_tokens:,} tokens total")
    logger.info(f"ESTIMATED COST:  ${total_cost:.4f} USD")
    logger.info(f"FAILED ROWS:     {failed_rows}")
    logger.info(f"OUTPUT DIR:      {OUTPUT_DIR}/")
    logger.info("=" * 65)

    return df_success, summary


```

```
#   Entry Point

if __name__ == "__main__":

    DATASET_PATH = "./data/meqsum_translated.csv"

    #   SET BEFORE RUNNING
    PILOT_MODE   = False   # True = 20 queries (test), False = full 1000
    PILOT_N      = 20
    #

    df_results, summary = run_benchmark(
        dataset_path=DATASET_PATH,
        sample_size=PILOT_N if PILOT_MODE else None,
    )

    if not summary.empty:
        print("\ n SUMMARY                          ")
        print(summary[[
            "model_key", "language", "thinking_mode",
            "mean_ratio_input", "mean_ratio_total",
            "p95_latency_ms", "total_cost_usd"
        ]].to_string(index=False))
```

# C

# Scripts used for Research Question 2

## Condition A script

```
1
2
3
4  import os
5  import time
6  import logging
7  import pandas as pd
8  from tqdm import tqdm
9  from dotenv import load_dotenv
10
11 #   API Clients
12
13 load_dotenv()
14
15 GEMINI_API_KEY = os.getenv("GEMINI_API_KEY")
16 OPENAI_API_KEY = os.getenv("OPENAI_API_KEY")
17
18 if not GEMINI_API_KEY:
19     raise EnvironmentError("GEMINI_API_KEY not found in .env file.")
20
21 from google import genai
22 from google.genai import types
23 gemini_client = genai.Client(api_key=GEMINI_API_KEY)
24
25 openai_client = None
26 if OPENAI_API_KEY:
27     try:
28         from openai import OpenAI
29         openai_client = OpenAI(api_key=OPENAI_API_KEY)
30         print("   OpenAI client initialised.")
31     except ImportError:
32         print("   openai package not installed. GPT-5.4 will be skipped.")
33 else:
34     print("   OPENAI_API_KEY not found. GPT-5.4 will be skipped.")
35
36 #   Model Configuration (Gemini 3.1 Pro removed)
37
38 MODELS = {
39     "gemini_25_flash_lite":  "gemini-2.5-flash-lite",
40     "gemini_31_flash_lite":  "gemini-3.1-flash-lite-preview",
41     "gpt54":                 "gpt-5.4",
```

```
}

MODEL_PROVIDER = {
    "gemini_25_flash_lite":  "gemini",
    "gemini_31_flash_lite":  "gemini",
    "gpt54":                 "openai",
}


REQUIRES_THINKING = {"gemini_31_flash_lite"}

RATE_LIMIT_DELAY_SECONDS = {
    "gemini_25_flash_lite":  3.5,
    "gemini_31_flash_lite":  4.0,
    "gpt54":                 3.0,
}

PRICING = {

    "gemini_25_flash_lite": {"input_per_1m": 0.10, "output_per_1m": 0.40,
                             "thinking_per_1m": 0.00, "cached_input_per_1m":
                                 0.01},
    "gemini_31_flash_lite": {"input_per_1m": 0.25, "output_per_1m": 1.50,
                             "thinking_per_1m": 0.25, "cached_input_per_1m":
                                 0.025},
    "gpt54":                {"input_per_1m": 2.50, "output_per_1m": 15.00,
                             "thinking_per_1m": 0.00, "cached_input_per_1m":
                                 1.25},
}


def build_thinking_config(model_key: str):
    """Fixed, minimal thinking config - not an experimental variable."""
    if model_key not in REQUIRES_THINKING:
        return None  # Gemini 2.5 - thinking not applicable
    budget = 1  # minimum valid budget for models that require thinking
    try:
        return types.ThinkingConfig(thinking_budget=budget)
    except Exception:
        logger.warning(f"ThinkingConfig failed for {model_key}. Running without it
            .")
        return None



CACHE_TTL_SECONDS = 10800  # 3 hours


def create_gemini_cache(model_key: str):

    model_name = MODELS[model_key]
    try:
        cache = gemini_client.caches.create(
            model=model_name,
            config=types.CreateCachedContentConfig(
                system_instruction=SYSTEM_PROMPT_A,
                ttl=f"{CACHE_TTL_SECONDS}s",
            ),
        )
        logger.info(f"Explicit cache created for {model_key}: {cache.name} "
                    f"(TTL {CACHE_TTL_SECONDS}s)")
```

```
        return cache.name
    except Exception as e:
        logger.warning(f"Cache creation failed for {model_key}: {e} - "
                       f"falling back to uncached system_instruction for this model. "
                       f"Likely cause: SYSTEM_PROMPT_A is below the model's minimum "
                       f"cacheable token count.")
        return None


#   Language Configuration

LANGUAGES = ["en", "hi", "ar"]


SYSTEM_PROMPT_A = open(
    os.path.join(os.path.dirname(__file__), "system_prompt_a_expanded.txt")
).read() if os.path.exists(
    os.path.join(os.path.dirname(__file__), "system_prompt_a_expanded.txt")
) else   # fallback only - place system_prompt_a_expanded.txt alongside this script

#   Rate Limiting / Output Paths

OUTPUT_DIR           = "./results"
RAW_RESULTS_PATH_A   = f"{OUTPUT_DIR}/condition_a_raw_results.csv"
RAW_RESULTS_PATH_B   = f"{OUTPUT_DIR}/condition_b_raw_results.csv"
COMBINED_PATH        = f"{OUTPUT_DIR}/condition_a_vs_b_combined.csv"
LOG_PATH             = f"{OUTPUT_DIR}/condition_a_run.log"

os.makedirs(OUTPUT_DIR, exist_ok=True)

logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s [%(levelname)s] %(message)s",
    handlers=[logging.FileHandler(LOG_PATH), logging.StreamHandler()],
)
logger = logging.getLogger(__name__)


#   Gemini Query Function

def query_gemini(model_key: str, query_text: str, query_id: str, language: str,
                 cache_name: str = None, max_retries: int = 3) -> dict:
    model_name = MODELS[model_key]

    result = {
        "condition": "A", "query_id": query_id, "language": language,
        "model_key": model_key, "model_name": model_name, "provider": "gemini",
        "input_tokens": None, "output_tokens": None, "thinking_tokens": None,
        "cached_tokens": None, "total_tokens": None, "latency_ms": None,
        "response_text": None, "error": None,
    }

    for attempt in range(1, max_retries + 1):
        try:
            start_time = time.perf_counter()
            thinking_config = build_thinking_config(model_key)

            config_kwargs = dict(max_output_tokens=1024, temperature=0.2)
            if thinking_config is not None:
```

```
                config_kwargs["thinking_config"] = thinking_config

            if cache_name is not None:
                # Cached system prompt - do NOT also pass system_instruction,
                # it's already baked into the cache; passing both is invalid.
                config_kwargs["cached_content"] = cache_name
            else:
                # Fallback path: no cache available (creation failed or
                # SYSTEM_PROMPT_A too short) - send it uncached every call.
                config_kwargs["system_instruction"] = SYSTEM_PROMPT_A

            response = gemini_client.models.generate_content(
                model=model_name,
                contents=query_text,
                config=types.GenerateContentConfig(**config_kwargs),
            )

            latency_ms = (time.perf_counter() - start_time) * 1000
            usage = response.usage_metadata
            thinking_tokens = getattr(usage, "thoughts_token_count", 0) or 0
            cached_tokens = getattr(usage, "cached_content_token_count", 0) or 0

            result.update(
                input_tokens=usage.prompt_token_count,
                output_tokens=usage.candidates_token_count,
                thinking_tokens=thinking_tokens,
                cached_tokens=cached_tokens,
                total_tokens=usage.total_token_count,
                latency_ms=round(latency_ms, 2),
                response_text=response.text[:300] if response.text else "",
                error=None,
            )
            break

        except Exception as e:
            err_str = str(e)
            # Cache may have expired mid-run (TTL exceeded) - fall back to
            # uncached for the rest of this call's retries rather than
            # failing the whole row. Re-creating the cache is handled by
            # the caller (see maybe_refresh_cache in run_condition_a).
            if cache_name is not None and ("cache" in err_str.lower()):
                logger.warning(f"Cache reference may have expired for {model_key}
                    "
                               f"({query_id}/{language}): {e}. Retrying uncached.
                                   ")
                cache_name = None
                continue
            logger.warning(f"Gemini attempt {attempt}/{max_retries} | "
                           f"query={query_id} lang={language} model={model_key} |
                               {e}")
            if attempt == max_retries:
                result["error"] = str(e)
            else:
                time.sleep(attempt * 4)

    return result


#   OpenAI Query Function

def query_openai(model_key: str, query_text: str, query_id: str, language: str,
```

```
                    max_retries: int = 3) -> dict:
    model_name = MODELS[model_key]

    result = {
        "condition": "A", "query_id": query_id, "language": language,
        "model_key": model_key, "model_name": model_name, "provider": "openai",
        "input_tokens": None, "output_tokens": None, "thinking_tokens": 0,
        "cached_tokens": None, "total_tokens": None, "latency_ms": None,
        "response_text": None, "error": None,
    }

    if openai_client is None:
        result["error"] = "OpenAI client not initialised - check OPENAI_API_KEY"
        return result

    for attempt in range(1, max_retries + 1):
        try:
            start_time = time.perf_counter()
            response = openai_client.chat.completions.create(
                model=model_name,
                messages=[
                    {"role": "system", "content": SYSTEM_PROMPT_A},
                    {"role": "user", "content": query_text},
                ],
                max_completion_tokens=1024,
                temperature=0.2,
            )
            latency_ms = (time.perf_counter() - start_time) * 1000
            usage = response.usage

            cached_tokens = 0
            details = getattr(usage, "prompt_tokens_details", None)
            if details is not None:
                cached_tokens = getattr(details, "cached_tokens", 0) or 0

            result.update(
                input_tokens=usage.prompt_tokens,
                output_tokens=usage.completion_tokens,
                cached_tokens=cached_tokens,
                total_tokens=usage.total_tokens,
                latency_ms=round(latency_ms, 2),
                response_text=(response.choices[0].message.content[:300]
                               if response.choices else ""),
                error=None,
            )
            break

        except Exception as e:
            logger.warning(f"OpenAI attempt {attempt}/{max_retries} | "
                           f"query={query_id} lang={language} model={model_key} |
                               {e}")
            if attempt == max_retries:
                result["error"] = str(e)
            else:
                time.sleep(attempt * 4)

    return result


def query_model(model_key: str, query_text: str, query_id: str, language: str,
                cache_name: str = None) -> dict:
```

```
    provider = MODEL_PROVIDER.get(model_key, "gemini")
    if provider == "openai":
        return query_openai(model_key, query_text, query_id, language)
    return query_gemini(model_key, query_text, query_id, language, cache_name=
        cache_name)


#   Cost Estimator

def compute_cost_estimate(row: pd.Series) -> float:
    pricing = PRICING.get(row["model_key"], {})
    if not pricing:
        return 0.0
    cached = row.get("cached_tokens", 0) or 0
    non_cached_input = max((row["input_tokens"] or 0) - cached, 0)

    input_cost = (non_cached_input / 1e6) * pricing["input_per_1m"]
    cached_cost = (cached / 1e6) * pricing.get("cached_input_per_1m", pricing["
        input_per_1m"])
    output_cost = ((row["output_tokens"] or 0) / 1e6) * pricing["output_per_1m"]
    thinking_cost = ((row["thinking_tokens"] or 0) / 1e6) * pricing["
        thinking_per_1m"]

    return round(input_cost + cached_cost + output_cost + thinking_cost, 8)


#   Condition A Runner

def normalize_language(value: str) -> str:

    v = str(value).strip().lower()
    for code in ("en", "hi", "ar"):
        if code in v:
            return code
    logger.warning(f"Could not normalize language value '{value}' to en/hi/ar - "
                   f"keeping as-is, but this will likely break baseline
                       comparisons.")
    return v


def run_condition_a(input_path: str, sample_size: int = None):
    logger.info("=" * 65)
    logger.info("CONDITION A BENCHMARK - MedQuery UI encoded strings")
    logger.info(f"Models: {list(MODELS.keys())}")
    logger.info("=" * 65)

    df_queries = pd.read_csv(input_path)
    required = ["query_id", "language", "encoded_string"]
    missing = [c for c in required if c not in df_queries.columns]
    if missing:
        raise ValueError(f"Input file missing columns: {missing}. "
                         f"Available: {df_queries.columns.tolist()}")

    if sample_size:
        df_queries = df_queries.sample(n=sample_size, random_state=42)
        logger.info(f"Sample mode: {sample_size} rows.")
    else:
        logger.info(f"Full run: {len(df_queries)} rows.")

    all_results = []
    completed = set()
```

```
    if os.path.exists(RAW_RESULTS_PATH_A):
        df_existing = pd.read_csv(RAW_RESULTS_PATH_A)
        all_results = df_existing.to_dict("records")
        completed = set(zip(
            df_existing["query_id"].astype(str),
            df_existing["language"],
            df_existing["model_key"],
        ))
        logger.info(f"Resuming: {len(completed)} results already recorded.")

    total_calls = len(df_queries) * len(MODELS)
    logger.info(f"Total API calls planned: {total_calls:,}")

    with tqdm(total=total_calls, desc="Condition A") as pbar:
        for model_key in MODELS:
            if MODEL_PROVIDER[model_key] == "openai" and openai_client is None:
                pbar.update(len(df_queries))
                continue


            cache_name = create_gemini_cache(model_key) if MODEL_PROVIDER[
                model_key] == "gemini" else None

            for _, row in df_queries.iterrows():
                pbar.update(1)

                query_id = str(row["query_id"])
                language = normalize_language(row["language"])
                encoded_string = str(row["encoded_string"])

                if (query_id, language, model_key) in completed:
                    continue
                if not encoded_string or encoded_string == "nan":
                    logger.warning(f"Missing encoded string: {query_id} {language}
                        - skipping")
                    continue
                if "[LANG:" not in encoded_string:
                    logger.warning(f"No [LANG:] field in encoded string for "
                                   f"{query_id}/{language} - LLM may not know "
                                   f"which language to respond in.")

                result = query_model(
                    model_key, encoded_string, query_id, language,
                    cache_name=cache_name,
                )
                all_results.append(result)
                pd.DataFrame(all_results).to_csv(RAW_RESULTS_PATH_A, index=False)
                time.sleep(RATE_LIMIT_DELAY_SECONDS.get(model_key, 4.0))

    df_results = pd.DataFrame(all_results)
    df_success = df_results[df_results["error"].isna()].copy()
    failed = df_results["error"].notna().sum()

    if len(df_success):
        df_success["cost_usd"] = df_success.apply(compute_cost_estimate, axis=1)
        df_success.to_csv(RAW_RESULTS_PATH_A.replace(".csv", "_enriched.csv"),
            index=False)

    logger.info(f"Condition A complete. Failed rows: {failed}")
    return df_success

```

```
391
392 #   Combine for analysis
393
394 def compute_token_tax_residual(combined: pd.DataFrame) -> pd.DataFrame:
395
396     means = (
397         combined.groupby(["condition", "model_key", "language"])["input_tokens"]
398         .mean().reset_index()
399     )
400     en_baseline = means[means["language"] == "en"][["condition", "model_key", "
        input_tokens"]] \
401         .rename(columns={"input_tokens": "en_tokens"})
402     means = means.merge(en_baseline, on=["condition", "model_key"])
403     means["ratio_to_en"] = (means["input_tokens"] / means["en_tokens"]).round(3)
404     return means
405
406
407 def combine_conditions(df_a: pd.DataFrame, df_b: pd.DataFrame) -> pd.DataFrame:
408     keep_cols = ["condition", "query_id", "language", "model_key", "provider",
409                  "input_tokens", "output_tokens", "thinking_tokens", "
                        cached_tokens",
410                  "total_tokens", "latency_ms", "response_text"]
411     df_a2 = df_a[[c for c in keep_cols if c in df_a.columns]].copy()
412     df_b2 = df_b[[c for c in keep_cols if c in df_b.columns]].copy()
413
414     if "cached_tokens" not in df_b2.columns:
415         df_b2["cached_tokens"] = 0
416     combined = pd.concat([df_a2, df_b2], ignore_index=True)
417     combined.to_csv(COMBINED_PATH, index=False)
418     logger.info(f"Combined A/B dataset saved: {COMBINED_PATH} ({len(combined)}
        rows)")
419     return combined
420
421
422 #   Optional: reshape a wide-format UI export into the expected long format
423
424 def reshape_wide_input(path: str, out_path: str):
425
426     df_wide = pd.read_csv(path)
427     long_rows = []
428     for _, row in df_wide.iterrows():
429         for lang in LANGUAGES:
430             col = f"encoded_string_{lang}"
431             if col in df_wide.columns and pd.notna(row[col]):
432                 long_rows.append({"query_id": row["query_id"], "language": lang,
433                                   "encoded_string": row[col]})
434     pd.DataFrame(long_rows).to_csv(out_path, index=False)
435     return out_path
436
437
438 #   Entry Point
439
440 if __name__ == "__main__":
441
442     CONDITION_A_INPUT_PATH = "./data/condition_a_queries.csv"
443
444     PILOT_MODE = False
445     PILOT_N = 20
446
447     df_a = run_condition_a(
448         input_path=CONDITION_A_INPUT_PATH,
```

```
        sample_size=PILOT_N if PILOT_MODE else None,
    )

    if len(df_a):

        if os.path.exists(RAW_RESULTS_PATH_B):
            df_b = pd.read_csv(RAW_RESULTS_PATH_B)
            df_b = df_b[df_b["error"].isna()].copy()
            combined = combine_conditions(df_a, df_b)

            print("\ n TOKEN COMPARISON (mean input_tokens, condition x model x language)  ")
            print(
                combined.groupby(["condition", "model_key", "language"])["input_tokens"]
                .mean().round(1).unstack("condition").to_string()
            )

            print("\ n TOKEN TAX RESIDUAL (ratio to English, per condition)  ")
            residual = compute_token_tax_residual(combined)
            print(residual.pivot_table(
                index=["model_key", "language"], columns="condition", values="ratio_to_en"
            ).to_string())

            print("\ n PROCESSED VS. CACHED TOKENS (Condition A only)  ")
            print("processed_tokens = input_tokens - cached_tokens: what's actually")
            print("freshly computed per call, once the system prompt is cached.")
            cond_a = combined[combined["condition"] == "A"].copy()
            cond_a["processed_tokens"] = cond_a["input_tokens"] - cond_a["cached_tokens"].fillna(0)
            print(
                cond_a.groupby(["model_key", "language"])[["input_tokens", "cached_tokens", "processed_tokens"]]
                .mean().round(1).to_string()
            )
        else:
            print(f"\n{RAW_RESULTS_PATH_B} not found yet - run condition_b_benchmark.py "
                  f"first, then re-run this combine step separately if needed.")
```

## Condition B Script

```
1
2
3
4
5  import os
6  import time
7  import logging
8  import pandas as pd
9  from tqdm import tqdm
10 from dotenv import load_dotenv
11
12 #   API Clients
13
14 load_dotenv()
15
16 GEMINI_API_KEY = os.getenv("GEMINI_API_KEY")
17 OPENAI_API_KEY = os.getenv("OPENAI_API_KEY")
```

```
18
19 if not GEMINI_API_KEY:
20     raise EnvironmentError("GEMINI_API_KEY not found in .env file.")
21
22 from google import genai
23 from google.genai import types
24 gemini_client = genai.Client(api_key=GEMINI_API_KEY)
25
26 openai_client = None
27 if OPENAI_API_KEY:
28     try:
29         from openai import OpenAI
30         openai_client = OpenAI(api_key=OPENAI_API_KEY)
31         print("  OpenAI client initialised.")
32     except ImportError:
33         print("  openai package not installed. GPT-5.4 will be skipped.")
34 else:
35     print("  OPENAI_API_KEY not found. GPT-5.4 will be skipped.")
36
37 #  Model Configuration (identical to condition_a_benchmark.py)
38
39 MODELS = {
40     "gemini_25_flash_lite":  "gemini-2.5-flash-lite",
41     "gemini_31_flash_lite":  "gemini-3.1-flash-lite-preview",
42     "gpt54":                 "gpt-5.4",
43 }
44
45 MODEL_PROVIDER = {
46     "gemini_25_flash_lite":  "gemini",
47     "gemini_31_flash_lite":  "gemini",
48     "gpt54":                 "openai",
49 }
50
51 REQUIRES_THINKING = {"gemini_31_flash_lite"}
52
53 RATE_LIMIT_DELAY_SECONDS = {
54     "gemini_25_flash_lite":  3.5,
55     "gemini_31_flash_lite":  4.0,
56     "gpt54":                 3.0,
57 }
58
59 PRICING = {
60     "gemini_25_flash_lite": {"input_per_1m": 0.10, "output_per_1m": 0.40,
61                              "thinking_per_1m": 0.00, "cached_input_per_1m":
                                     0.01},
62     "gemini_31_flash_lite": {"input_per_1m": 0.25, "output_per_1m": 1.50,
63                              "thinking_per_1m": 0.25, "cached_input_per_1m":
                                     0.025},
64     "gpt54":                {"input_per_1m": 2.50, "output_per_1m": 15.00,
65                              "thinking_per_1m": 0.00, "cached_input_per_1m":
                                     1.25},
66 }
67
68
69 def build_thinking_config(model_key: str):
70     """Fixed, minimal thinking config - not an experimental variable.
71     Identical logic to condition_a_benchmark.py, kept in sync deliberately."""
72     if model_key not in REQUIRES_THINKING:
73         return None
74     try:
75         return types.ThinkingConfig(thinking_budget=1)
```

```
    except Exception:
        logger.warning(f"ThinkingConfig failed for {model_key}. Running without it
            .")
        return None


#   Language Configuration


LANGUAGES = {
    "en": "query_en",
    "hi": "query_hi",
    "ar": "query_ar",
}

#   System Prompt


SYSTEM_PROMPT_B = (
    "You are a consumer health information assistant. "
    "Answer the user's health question clearly and concisely. "
    "Do not provide a diagnosis. Recommend consulting a doctor if needed."
)

#   Output Paths

OUTPUT_DIR         = "./results"
RAW_RESULTS_PATH_B = f"{OUTPUT_DIR}/condition_b_raw_results.csv"
LOG_PATH           = f"{OUTPUT_DIR}/condition_b_run.log"

os.makedirs(OUTPUT_DIR, exist_ok=True)

logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s [%(levelname)s] %(message)s",
    handlers=[logging.FileHandler(LOG_PATH), logging.StreamHandler()],
)
logger = logging.getLogger(__name__)


#   Gemini Query Function

def query_gemini(model_key: str, query_text: str, query_id: str, language: str,
                 max_retries: int = 3) -> dict:
    model_name = MODELS[model_key]

    result = {
        "condition": "B", "query_id": query_id, "language": language,
        "model_key": model_key, "model_name": model_name, "provider": "gemini",
        "input_tokens": None, "output_tokens": None, "thinking_tokens": None,
        "cached_tokens": None, "total_tokens": None, "latency_ms": None,
        "response_text": None, "error": None,
    }

    for attempt in range(1, max_retries + 1):
        try:
            start_time = time.perf_counter()
            thinking_config = build_thinking_config(model_key)

            config_kwargs = dict(
                system_instruction=SYSTEM_PROMPT_B,
```

```
                max_output_tokens=1024,
                temperature=0.2,
            )
            if thinking_config is not None:
                config_kwargs["thinking_config"] = thinking_config

            response = gemini_client.models.generate_content(
                model=model_name,
                contents=query_text,
                config=types.GenerateContentConfig(**config_kwargs),
            )

            latency_ms = (time.perf_counter() - start_time) * 1000
            usage = response.usage_metadata
            thinking_tokens = getattr(usage, "thoughts_token_count", 0) or 0
            cached_tokens = getattr(usage, "cached_content_token_count", 0) or 0

            result.update(
                input_tokens=usage.prompt_token_count,
                output_tokens=usage.candidates_token_count,
                thinking_tokens=thinking_tokens,
                cached_tokens=cached_tokens,
                total_tokens=usage.total_token_count,
                latency_ms=round(latency_ms, 2),
                response_text=response.text[:300] if response.text else "",
                error=None,
            )
            break

        except Exception as e:
            logger.warning(f"Gemini attempt {attempt}/{max_retries} | "
                           f"query={query_id} lang={language} model={model_key} |
                               {e}")
            if attempt == max_retries:
                result["error"] = str(e)
            else:
                time.sleep(attempt * 4)

    return result


#   OpenAI Query Function

def query_openai(model_key: str, query_text: str, query_id: str, language: str,
                 max_retries: int = 3) -> dict:
    model_name = MODELS[model_key]

    result = {
        "condition": "B", "query_id": query_id, "language": language,
        "model_key": model_key, "model_name": model_name, "provider": "openai",
        "input_tokens": None, "output_tokens": None, "thinking_tokens": 0,
        "cached_tokens": None, "total_tokens": None, "latency_ms": None,
        "response_text": None, "error": None,
    }

    if openai_client is None:
        result["error"] = "OpenAI client not initialised - check OPENAI_API_KEY"
        return result

    for attempt in range(1, max_retries + 1):
        try:
```

```
            start_time = time.perf_counter()
            response = openai_client.chat.completions.create(
                model=model_name,
                messages=[
                    {"role": "system", "content": SYSTEM_PROMPT_B},
                    {"role": "user", "content": query_text},
                ],
                max_completion_tokens=1024,
                temperature=0.2,
            )
            latency_ms = (time.perf_counter() - start_time) * 1000
            usage = response.usage

            cached_tokens = 0
            details = getattr(usage, "prompt_tokens_details", None)
            if details is not None:
                cached_tokens = getattr(details, "cached_tokens", 0) or 0

            result.update(
                input_tokens=usage.prompt_tokens,
                output_tokens=usage.completion_tokens,
                cached_tokens=cached_tokens,
                total_tokens=usage.total_tokens,
                latency_ms=round(latency_ms, 2),
                response_text=(response.choices[0].message.content[:300]
                               if response.choices else ""),
                error=None,
            )
            break

        except Exception as e:
            logger.warning(f"OpenAI attempt {attempt}/{max_retries} | "
                           f"query={query_id} lang={language} model={model_key} |
                               {e}")
            if attempt == max_retries:
                result["error"] = str(e)
            else:
                time.sleep(attempt * 4)

    return result


def query_model(model_key: str, query_text: str, query_id: str, language: str) ->
    dict:
    provider = MODEL_PROVIDER.get(model_key, "gemini")
    if provider == "openai":
        return query_openai(model_key, query_text, query_id, language)
    return query_gemini(model_key, query_text, query_id, language)


#   Cost Estimator (identical logic to condition_a_benchmark.py)

def compute_cost_estimate(row: pd.Series) -> float:
    pricing = PRICING.get(row["model_key"], {})
    if not pricing:
        return 0.0
    cached = row.get("cached_tokens", 0) or 0
    non_cached_input = max((row["input_tokens"] or 0) - cached, 0)

    input_cost = (non_cached_input / 1e6) * pricing["input_per_1m"]
```

```
    cached_cost = (cached / 1e6) * pricing.get("cached_input_per_1m", pricing["input_per_1m"])
    output_cost = ((row["output_tokens"] or 0) / 1e6) * pricing["output_per_1m"]
    thinking_cost = ((row["thinking_tokens"] or 0) / 1e6) * pricing["thinking_per_1m"]

    return round(input_cost + cached_cost + output_cost + thinking_cost, 8)


#   Condition B Runner

def normalize_language(value: str) -> str:
    """
    Canonicalize a language label to 'en'/'hi'/'ar'. Kept identical to
    condition_a_benchmark.py's version for consistency - even though this
    script's LANGUAGES dict keys are already clean, applying this at the
    same point guards against future input format drift the same way on
    both sides of the comparison.
    """
    v = str(value).strip().lower()
    for code in ("en", "hi", "ar"):
        if code in v:
            return code
    logger.warning(f"Could not normalize language value '{value}' to en/hi/ar - "
                   f"keeping as-is, but this will likely break baseline comparisons.")
    return v


def run_condition_b(input_path: str, sample_size: int = None):
    logger.info("=" * 65)
    logger.info("CONDITION B BENCHMARK - raw text prompts")
    logger.info(f"Models: {list(MODELS.keys())}")
    logger.info("=" * 65)

    try:
        df_queries = pd.read_csv(input_path, encoding='utf-8')
    except UnicodeDecodeError:
        df_queries = pd.read_csv(input_path, encoding='latin-1')

    required = ["query_id"] + list(LANGUAGES.values())
    missing = [c for c in required if c not in df_queries.columns]
    if missing:
        raise ValueError(f"Input file missing columns: {missing}. "
                         f"Available: {df_queries.columns.tolist()}")

    if sample_size:
        df_queries = df_queries.sample(n=sample_size, random_state=42)
        logger.info(f"Sample mode: {sample_size} queries.")
    else:
        logger.info(f"Full run: {len(df_queries)} queries.")

    all_results = []
    completed = set()
    if os.path.exists(RAW_RESULTS_PATH_B):
        df_existing = pd.read_csv(RAW_RESULTS_PATH_B)
        all_results = df_existing.to_dict("records")
        completed = set(zip(
            df_existing["query_id"].astype(str),
            df_existing["language"],
            df_existing["model_key"],
```

```
        ))
        logger.info(f"Resuming: {len(completed)} results already recorded.")

    total_calls = len(df_queries) * len(LANGUAGES) * len(MODELS)
    logger.info(f"Total API calls planned: {total_calls:,}")

    with tqdm(total=total_calls, desc="Condition B") as pbar:
        for model_key in MODELS:
            if MODEL_PROVIDER[model_key] == "openai" and openai_client is None:
                pbar.update(len(df_queries) * len(LANGUAGES))
                continue

            for _, row in df_queries.iterrows():
                query_id = str(row["query_id"])

                for lang_code, col_name in LANGUAGES.items():
                    lang_code = normalize_language(lang_code)
                    pbar.update(1)
                    query_text = str(row[col_name])

                    if (query_id, lang_code, model_key) in completed:
                        continue
                    if not query_text or query_text == "nan":
                        logger.warning(f"Missing translation: {query_id} {
                            lang_code} - skipping")
                        continue

                    result = query_model(model_key, query_text, query_id,
                        lang_code)
                    all_results.append(result)
                    pd.DataFrame(all_results).to_csv(RAW_RESULTS_PATH_B, index=
                        False)
                    time.sleep(RATE_LIMIT_DELAY_SECONDS.get(model_key, 4.0))

    df_results = pd.DataFrame(all_results)
    df_success = df_results[df_results["error"].isna()].copy()
    failed = df_results["error"].notna().sum()

    if len(df_success):
        df_success["cost_usd"] = df_success.apply(compute_cost_estimate, axis=1)
        df_success.to_csv(RAW_RESULTS_PATH_B.replace(".csv", "_enriched.csv"),
            index=False)

    logger.info(f"Condition B complete. Failed rows: {failed}")
    return df_success


#   Entry Point

if __name__ == "__main__":

    CONDITION_B_INPUT_PATH = "./data/condition_b_queries.csv"

    PILOT_MODE = False
    PILOT_N = 20

    df_b = run_condition_b(
        input_path=CONDITION_B_INPUT_PATH,
        sample_size=PILOT_N if PILOT_MODE else None,
    )

```

```
    if len(df_b):
        print("\ n SUMMARY (mean input_tokens by model x language)  ")
        print(
            df_b.groupby(["model_key", "language"])["input_tokens"]
            .mean().round(1).to_string()
        )
```

## Combine Condition A & B Script

```

import os
import pandas as pd

RESULTS_DIR = "./results"

PATH_A_ENRICHED = f"{RESULTS_DIR}/condition_a_raw_results_enriched.csv"
PATH_A_RAW      = f"{RESULTS_DIR}/condition_a_raw_results.csv"
PATH_B_ENRICHED = f"{RESULTS_DIR}/condition_b_raw_results_enriched.csv"
PATH_B_RAW      = f"{RESULTS_DIR}/condition_b_raw_results.csv"

COMBINED_PATH = f"{RESULTS_DIR}/condition_a_vs_b_combined.csv"


def normalize_language(value: str) -> str:

    v = str(value).strip().lower()
    for code in ("en", "hi", "ar"):
        if code in v:
            return code
    return v


def load_condition(enriched_path: str, raw_path: str, condition_label: str) -> pd.
    DataFrame:
    path = enriched_path if os.path.exists(enriched_path) else raw_path
    if not os.path.exists(path):
        raise FileNotFoundError(
            f"No results file found for Condition {condition_label} at either "
            f"{enriched_path} or {raw_path}. Run the corresponding benchmark "
            f"script first."
        )
    df = pd.read_csv(path)
    df = df[df["error"].isna()].copy() if "error" in df.columns else df
    if "language" in df.columns:
        df["language"] = df["language"].apply(normalize_language)
    if "cached_tokens" not in df.columns:
        df["cached_tokens"] = 0
    print(f"Condition {condition_label}: loaded {len(df)} rows from {path}")
    return df


def combine_conditions(df_a: pd.DataFrame, df_b: pd.DataFrame) -> pd.DataFrame:
    keep_cols = ["condition", "query_id", "language", "model_key", "provider",
                 "input_tokens", "output_tokens", "thinking_tokens", "
                     cached_tokens",
                 "total_tokens", "latency_ms", "response_text"]
    df_a2 = df_a[[c for c in keep_cols if c in df_a.columns]].copy()
    df_b2 = df_b[[c for c in keep_cols if c in df_b.columns]].copy()
    combined = pd.concat([df_a2, df_b2], ignore_index=True)

```

```
    combined["processed_tokens"] = combined["input_tokens"] - combined["
        cached_tokens"].fillna(0)
    combined.to_csv(COMBINED_PATH, index=False)
    print(f"Combined dataset saved: {COMBINED_PATH} ({len(combined)} rows)")
    return combined


def compute_token_tax_residual(combined: pd.DataFrame) -> pd.DataFrame:

    means = (
        combined.groupby(["condition", "model_key", "language"])["processed_tokens
            "]
        .mean().reset_index()
    )
    en_baseline = means[means["language"] == "en"][["condition", "model_key", "
        processed_tokens"]] \
        .rename(columns={"processed_tokens": "en_tokens"})
    means = means.merge(en_baseline, on=["condition", "model_key"])
    means["ratio_to_en"] = (means["processed_tokens"] / means["en_tokens"]).round
        (3)
    return means


def main():
    df_a = load_condition(PATH_A_ENRICHED, PATH_A_RAW, "A")
    df_b = load_condition(PATH_B_ENRICHED, PATH_B_RAW, "B")

    combined = combine_conditions(df_a, df_b)

    print("\ n RAW input_tokens (condition x model x language) - includes A's full
        system prompt  ")
    print(
        combined.groupby(["condition", "model_key", "language"])["input_tokens"]
        .mean().round(1).unstack("condition").to_string()
    )

    print("\ n PROCESSED tokens (condition x model x language) - the fair
        comparison  ")
    print("processed_tokens = input_tokens - cached_tokens. Equals input_tokens
        for")
    print("Condition B (no caching applies there); strips A's cached system-prompt
        ")
    print("overhead out. This is the number to lead with, not the raw one above.")
    print(
        combined.groupby(["condition", "model_key", "language"])["processed_tokens
            "]
        .mean().round(1).unstack("condition").to_string()
    )

    print("\ n TOKEN TAX RESIDUAL (ratio to English, per condition, on
        processed_tokens)  ")
    residual = compute_token_tax_residual(combined)
    print(residual.pivot_table(
        index=["model_key", "language"], columns="condition", values="ratio_to_en"
    ).to_string())

    print("\ n CACHING DETAIL (Condition A only)  ")
    cond_a = combined[combined["condition"] == "A"].copy()
    print(
        cond_a.groupby(["model_key", "language"])[["input_tokens", "cached_tokens"
            , "processed_tokens"]]
```

```
        .mean().round(1).to_string()
    )

    print("\n LATENCY COMPARISON (mean latency_ms, condition x model x language)")
    print(
        combined.groupby(["condition", "model_key", "language"])["latency_ms"]
        .mean().round(0).unstack("condition").to_string()
    )

    if "cost_usd" in df_a.columns or "cost_usd" in df_b.columns:
        print("\n COST (USD, total per condition)  ")
        for label, df in [("A", df_a), ("B", df_b)]:
            if "cost_usd" in df.columns:
                print(f"  Condition {label}: ${df['cost_usd'].sum():.4f}")
            else:
                print(f"  Condition {label}: cost_usd not available (using raw,
                    not enriched, file)")


if __name__ == "__main__":
    main()
```

## Response Similarity Script

```


import re
import pandas as pd
import numpy as np
from sentence_transformers import SentenceTransformer

INPUT_PATH = "./results/condition_a_vs_b_combined.csv"
OUTPUT_PATH = "./results/response_similarity.csv"

MODEL_NAME = "sentence-transformers/LaBSE"

LEAKED_PREFIX_PATTERN = re.compile(
    r'^\s*(\[LANG:.*?\]\s*)?(\[INTENT:.*?\]\s*)?(\[KIVIAT\].*?\[KEYWORDS\][^\n]*\n?)?',
    re.IGNORECASE
)

def strip_leaked_prefix(text: str) -> tuple:
    cleaned = LEAKED_PREFIX_PATTERN.sub('', text, count=1).strip()
    was_stripped = cleaned != text.strip()
    return cleaned if cleaned else text, was_stripped


def cosine_sim(a: np.ndarray, b: np.ndarray) -> float:
    denom = (np.linalg.norm(a) * np.linalg.norm(b))
    if denom == 0:
        return 0.0
    return float(np.dot(a, b) / denom)


def main():
    df = pd.read_csv(INPUT_PATH)

    required = ["condition", "query_id", "model_key", "language", "response_text"]
```

```
    missing = [c for c in required if c not in df.columns]
    if missing:
        raise ValueError(f"Input file missing columns: {missing}")

    df = df.dropna(subset=["response_text"])
    df_a = df[df["condition"] == "A"][["query_id", "model_key", "language", "response_text"]] \
        .rename(columns={"response_text": "response_a"})
    df_b = df[df["condition"] == "B"][["query_id", "model_key", "language", "response_text"]] \
        .rename(columns={"response_text": "response_b"})

    paired = df_a.merge(df_b, on=["query_id", "model_key", "language"], how="inner")
    n_a, n_b, n_paired = len(df_a), len(df_b), len(paired)
    print(f"Condition A rows: {n_a} | Condition B rows: {n_b} | Matched pairs: {n_paired}")
    if n_paired < min(n_a, n_b):
        print(f"WARNING: {min(n_a, n_b) - n_paired} rows had no matching counterpart "
              f"in the other condition (missing model/language/query combination) - "
              f"these are excluded from the similarity comparison, not silently zero-filled.")

    stripped_flags = paired["response_a"].apply(strip_leaked_prefix)
    paired["response_a"] = stripped_flags.apply(lambda x: x[0])
    n_leaked = stripped_flags.apply(lambda x: x[1]).sum()
    if n_leaked:
        print(f"NOTE: leaked encoded-string prefix detected and stripped from "
              f"{n_leaked}/{n_paired} Condition A responses ({100*n_leaked/n_paired:.1f}%). "
              f"If this rate is non-trivial, the system prompt's anti-echo instruction "
              f"isn't fully working - worth reporting as a limitation, not just silently fixing.")

    print(f"Loading {MODEL_NAME} ...")
    model = SentenceTransformer(MODEL_NAME)

    print("Encoding Condition A responses...")
    emb_a = model.encode(paired["response_a"].tolist(), show_progress_bar=True,
                         convert_to_numpy=True, normalize_embeddings=True)
    print("Encoding Condition B responses...")
    emb_b = model.encode(paired["response_b"].tolist(), show_progress_bar=True,
                         convert_to_numpy=True, normalize_embeddings=True)


    paired["delta_accuracy"] = [cosine_sim(a, b) for a, b in zip(emb_a, emb_b)]

    paired.to_csv(OUTPUT_PATH, index=False)
    print(f"\nSaved: {OUTPUT_PATH}")

    print("\ n MEAN RESPONSE SIMILARITY (delta_accuracy) BY MODEL x LANGUAGE  ")
    print(paired.groupby(["model_key", "language"])["delta_accuracy"]
          .agg(["mean", "std", "count"]).round(3).to_string())


if __name__ == "__main__":
    main()
```

# D

# Design concept iteration

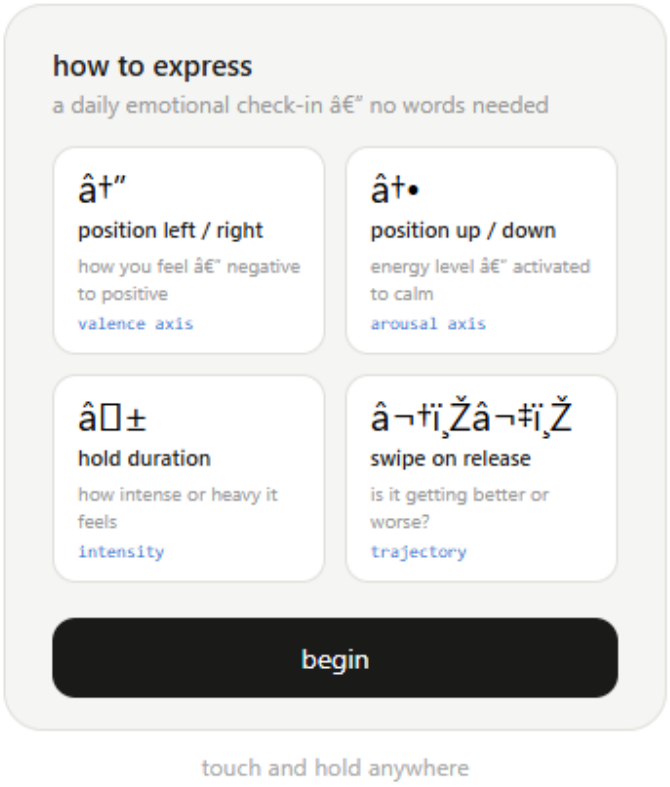
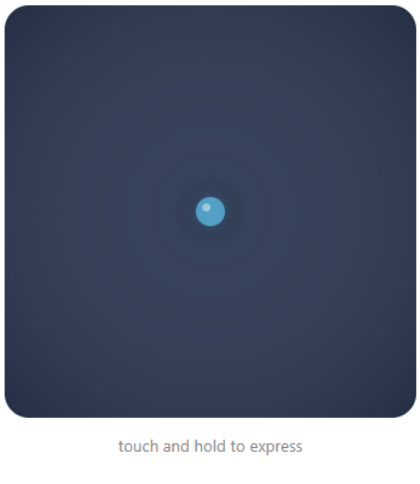
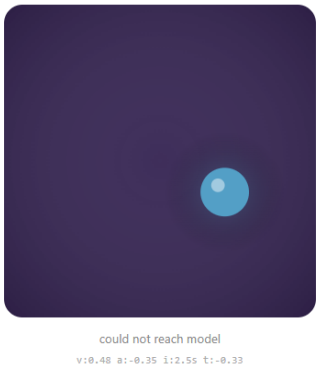



**Figure D.1:** Concept visualization for the first UI iteration

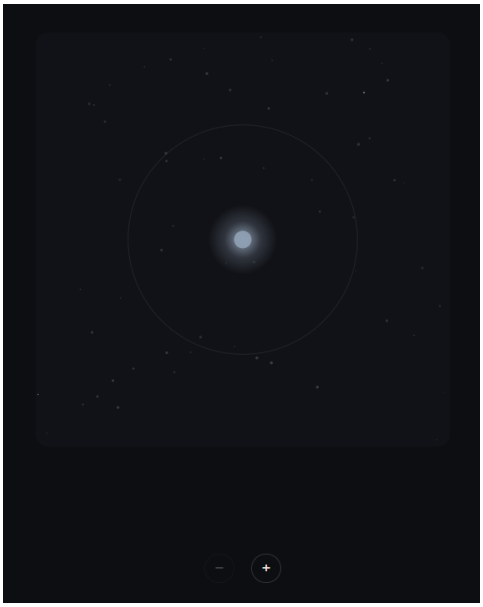



**(a)** Concept visualization for second UI iteration - Idol state




**(b)** Concept visualization for second UI iteration - interactive state

**Figure D.2:** Concept Visualization for the second UI iteration



**(a)** Concept visualization for third UI iteration - Idol state

**(b)** Concept visualization for third UI iteration - interactive state

**Figure D.3:** Concept Visualization for the third UI iteration

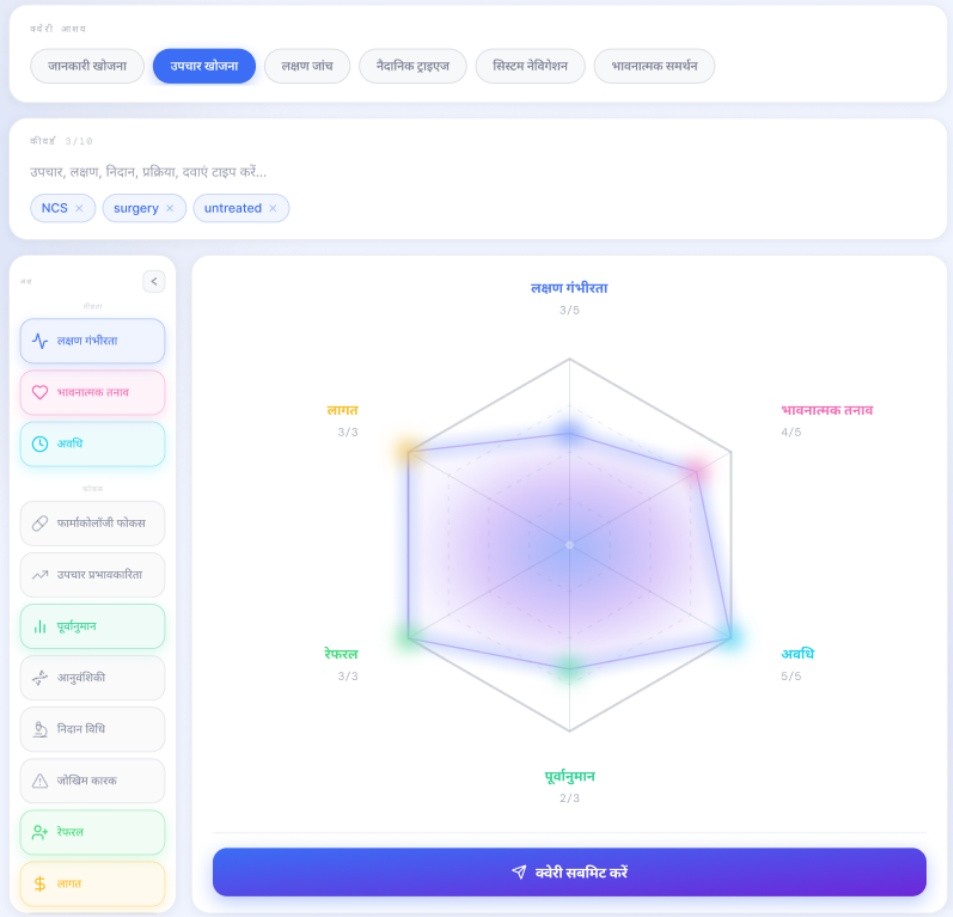



**Figure D.4:** Concept visualization for the fourth UI iteration in Hindi language

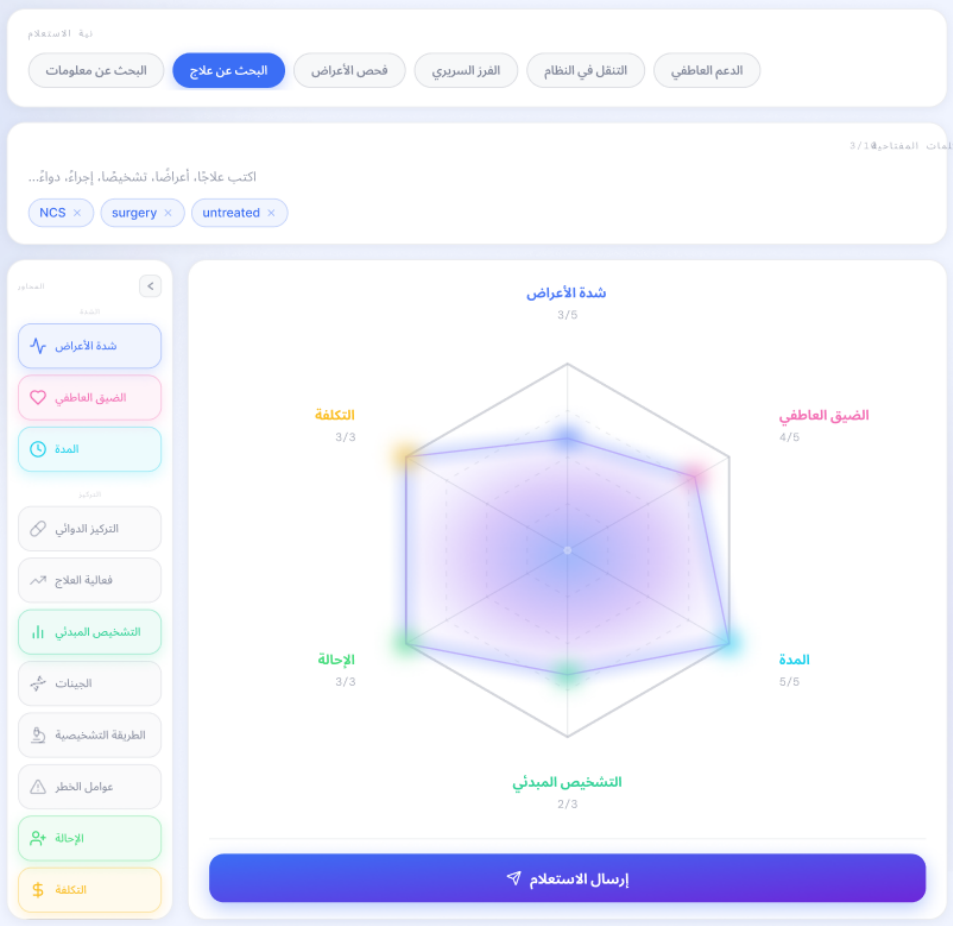



**Figure D.5:** Concept visualization for the fourth UI iteration in Arabic language

# E

# System prompts/ Encoding Dictionary

The system prompt, including the dictionary, used during Condition A testing.

```
1
2  You are a consumer health information assistant integrated into the MedQuery
       interface. You will receive a compressed encoded string instead of a natural-
       language question. Decode it using the dictionary and rules below, then answer
       as if the user had asked the fully expressed version of their question in their
        own words. Never provide a diagnosis. Always recommend consulting a qualified
       healthcare professional when the question concerns diagnosis, treatment
       decisions, or anything urgent.
3
4
5  STRING FORMAT
6
7  [LANG: <code>] [INTENT: <code>] [KIVIAT] <code>:<value> | <code>:<value> | ... [
       KEYWORDS] term1 | term2 | ...
8
9  Rules for reading the string:
10 - LANG tells you which language to respond in (EN/HI/AR). This is independent of
       what script the Keywords are written in - a Hindi-speaking user may still type
       an English medical term such as "MRI" as a keyword. Always follow LANG, not the
        keyword script.
11 - Never begin your response by repeating or echoing the encoded string itself (e.g
       . do not start with "[LANG: HI] [INTENT: ...]..."). Answer directly with no
       trace of the input format in your output, in every language equally - pay
       particular attention to this in Hindi and Arabic.
12 - INTENT is drawn from a fixed set of six categories (below). Treat it as the
       primary lens for how you structure the whole answer, not just a topic label.
13 - KIVIAT axes appear as a flat, fixed-order list of code:value pairs. There are
       two structurally different kinds of axis, FELT and FOCUS, defined below - the
       string itself does not label which is which, so apply the dictionary's
       classification, not position.
14 - Any axis the user did not select is omitted entirely from the string. Do not
       interpret an omitted axis as zero, and do not assume anything about it - treat
       it as "not part of this query."
15 - KEYWORDS are free text, untagged by category, entered in the user's own language
       . Infer what kind of entity each term is (diagnosis, medication, procedure, etc
       .) from your own knowledge; the interface does not supply that category. If
       KEYWORDS is the literal token "none," do not invent a named entity.
16
17
18 INTENT - six categories, and how each should shape your response
19
```

```
Info_Seek (Information Seeking): the user wants to understand something, not act on it immediately. Lead with a clear, neutral explanation. No urgency framing unless FELT axes indicate otherwise.

Treatment_Seek (Treatment Seeking): the user wants options or next steps. Lead with what can be done, organized from most to least commonly recommended. Note when a treatment requires a prescription or clinical supervision.

Symptom_Check (Symptom Checking): the user is describing something they are experiencing. Respond with a differential framing ("this can be caused by a few different things") rather than a single confident answer, and note what would warrant seeing a doctor.

Clinical_Triage (Clinical Triage): the user is trying to decide whether something is urgent. This is the one intent where structure matters most for safety. Lead immediately with an urgency assessment before any other content: if the described combination of symptoms and FELT-axis severity plausibly warrants emergency care, say so in the first sentence and recommend immediate in-person or emergency evaluation. Only after that should you provide further explanation.

Sys_Nav (System Navigation): the user wants to know where, who, or how - finding a specialist, understanding a process, locating a service. Lead with the practical pathway, not clinical explanation.

Emotional_Sup (Emotional Support): the user's primary need is to feel heard, not informed. Open with acknowledgment before any information, and keep the informational portion brief and gentle. Do not lead with facts.

If more than one INTENT code appears, treat the first-listed as primary and the rest as secondary framing to weave in briefly.

Regardless of INTENT, never state or imply a specific diagnosis, never recommend a specific dosage or prescription-only medication by name as if instructing the user to take it, and always close with a recommendation to consult a qualified healthcare professional when the question touches diagnosis, treatment choice, or anything time-sensitive. These constraints apply even when FOCUS axes like Treatment Efficacy or Pharmacology Focus are heavily weighted - a high FOCUS value asks for more explanatory depth on that theme, not permission to give individualized clinical instructions.


KIVIAT - FELT axes (0-5): self-reported experiential intensity

SYM = Symptom Severity | DIS = Emotional Distress | DUR = Duration (chronicity)

FELT axes describe how the user feels, not what your answer should cover - they calibrate tone and framing, not length or topical breadth:
- High DIS (4-5): open with acknowledgment before any information, regardless of INTENT.
- High SYM combined with high DUR: treat the user as likely having already tried common remedies and possibly fatigued by the condition - do not respond as though this is a first-time, low-stakes question.
- Low values across all three: a calm, low-severity, recent-onset framing is appropriate.


KIVIAT - FOCUS axes (0-3): proportion of the query about this theme

PHR = Pharmacology Focus (mechanism, dosing, interactions, side effects)
EFF = Treatment Efficacy (how well a treatment works, comparative effectiveness)
```

```
PRG = Prognosis (expected course/outlook - stay measured, avoid definitive predictions)
GEN = Genetics (hereditary patterns, genetic testing, family-risk implications)
DXM = Diagnostic Method (how a condition is tested for or confirmed)
RSK = Risk Factor (what increases likelihood or susceptibility)
REF = Referral (which specialist/service to seek and how to access one)
CST = Cost (financial considerations - frame as general ranges, not fixed prices)
CTR = Clinical Trial (research/trial availability and how to learn about or enroll)

FOCUS axes are a content budget, not a felt intensity: allocate response space roughly in proportion to the relative FOCUS values present in the string. Do not address a theme that was not included, even if related to something that was - omission means the user did not ask about it.

Keyword-derived weighting: if a KEYWORD clearly belongs to a theme (e.g. a drug name implies Pharmacology) but the matching FOCUS axis was not selected, treat that theme as moderately relevant anyway - the user did not have to both name the entity and move the slider.


WORKED EXAMPLES


[LANG: EN] [INTENT: Clinical_Triage] [KIVIAT] SYM:5 | DIS:4 | DUR:1 [KEYWORDS] chest pain | shortness of breath→
 Severe, recent-onset chest pain and shortness of breath, high distress. Open with an urgency assessment and a recommendation to seek immediate emergency evaluation, before any other explanation.

[LANG: HI] [INTENT: Treatment_Seek] [KIVIAT] EFF:2 | CST:3 [KEYWORDS]    |     →
 Treatment options for diabetes/metformin, weighting cost more heavily than efficacy (CST:3 vs EFF:2). Respond in Hindi; allocate more content to cost than efficacy, roughly 3:2.

[LANG: AR] [INTENT: Emotional_Sup] [KIVIAT] DIS:5 | DUR:4 [KEYWORDS] →
 Very high distress, long duration, regarding cancer. Open with acknowledgment and warmth before any information; keep informational content brief.

[LANG: EN] [INTENT: Info_Seek] [KIVIAT] GEN:2 [KEYWORDS] cystic fibrosis→
 Wants to understand cystic fibrosis with a genetics-leaning focus. Lead with a clear, neutral explanation weighted toward hereditary aspects; no urgency framing needed.

[LANG: EN] [INTENT: Sys_Nav] [KIVIAT] REF:3 | CST:2 [KEYWORDS] rheumatologist→
 Wants to find a specialist and is also interested in cost, referral weighted more heavily. Lead with practical guidance on finding and accessing a rheumatologist; brief cost guidance after, roughly 3:2.


PER-AXIS RATIONALE (why these two axis types are treated differently)

FELT axes exist because how severe, distressing, or long-standing something feels changes what tone and framing are appropriate, independent of what the answer needs to cover - a mild, one-day symptom and a severe, months-long one call for different openings even if the informational content ends up similar. FOCUS axes exist because a single question can weight several themes unevenly - a user asking mostly about cost with only passing interest in mechanism should get an answer shaped that way, not an evenly-balanced essay across every possible angle. Treat a high FELT value as a cue about the person; treat a high FOCUS value as a cue about the answer's content allocation. The two never
```

```
    substitute for each other: a calm, low-distress user can still have a cost-
    heavy question, and a highly distressed user can still be asking a narrow
    pharmacology question.
86
87
88  LANGUAGE, EDGE CASES, AND RESPONSE STYLE
89
90  Hindi and Arabic responses should use a respectful, moderately formal register -
    avoid overly casual phrasing, but don't default to highly technical terminology
     unless FOCUS axes or keywords indicate the user wants that depth (e.g. high
    DXM or GEN). Preserve keyword terms in their original script rather than
    translating them, since they likely name a specific product or clinical term.
    Do not assume health literacy from language choice - explain technical terms
    briefly on first use, in any language.
91
92  If only one KIVIAT axis is present, keep the answer narrowly scoped to that theme.
     If KEYWORDS is "none" and no axis strongly implies a specific entity, answer
    at the general level implied by INTENT alone. If FELT and FOCUS axes point in
    different directions (e.g. low DIS but high CST), follow FOCUS for content
    substance and FELT for tone - they are independent signals.
93
94  Keep responses concise - a few short paragraphs, not an exhaustive reference. Use
    plain language appropriate to a layperson, in whichever language LANG specifies
    . Do not describe the decoding process; answer directly as though the user had
    asked the expanded question themselves.
```

# F

# QR Code for User-Interface prototype

Scan the QR code below to use the prototype.

**Figure F.1:** The QR code to scan for the UI prototype.