ChatGPT

Toward N-Dimensional LLMs with Information Bottlenecks

Introduction

Transformer-based large language models (LLMs) have achieved remarkable performance in natural language tasks, but they face fundamental efficiency and grounding challenges. One key limitation is the **1-dimensional (1-D) token sequence architecture**: standard LLMs process long inputs as flat sequences of thousands of tokens, incurring quadratic attention costs and often losing focus on relevant details in lengthy contexts [1] [2]. Even with extended context windows up to hundreds of thousands of tokens, purely text-based models remain inefficient and can struggle with "lost in the middle" issues as irrelevant or redundant information dilutes their attention [1] [2]. This not only causes **latency and memory bottlenecks**, but also leaves models prone to **hallucination** when required information is missing or buried in the input. Achieving human-level intelligence will require absorbing and **grounding on vastly larger quantities of information** (text, visual, temporal, etc.) than current 1-D models can handle [3] [4].

Recent advances hint at a paradigm shift: instead of feeding an LLM massive raw text, we can **compress and enrich the input with multi-dimensional signals** to maximize relevant information per token. For example, the DeepSeek-OCR system represents long documents as images, achieving a ~10× reduction in token count with minimal loss of content [5]. By treating vision not as an add-on but as a **context compression layer**, DeepSeek-OCR showed that a page of text (normally 2000–5000 tokens) can be encoded in an image requiring only ~200–400 visual tokens [5]. The model's visual encoder and decoder then reconstruct the text with high fidelity even at 10× compression, retaining ~97% of words correctly [6]. This suggests that **2-dimensional (2-D) representations** (spatial layouts of text) can convey information much more efficiently than linear text, tackling the long-context problem at its root. Moreover, providing an LLM with non-textual modalities or metadata (images, layout, etc.) can reduce hallucinations by grounding the model in what it "sees" in the input [7].

In this whitepaper, we propose a comprehensive framework to generalize such approaches and push beyond 2-D to **N-dimensional (N-D) inputs**. We combine formal principles from information theory with new architecture designs to create **variable-rate token bottlenecks** and **synchronized multi-dimensional inputs**. The goal is to maximize the **mutual information** between input and output per token used, thus increasing the model's information efficiency and factual grounding as a step toward more general intelligence. We will:

- **Contextualize current 1-D vs. 2-D LLM architectures:** analyzing the limitations of linear token processing and reviewing new 2-D models (e.g. document vision-LMs like DeepSeek-OCR and LayoutLM) [8] [9]. We critique their strengths and shortcomings as a foundation for N-D generalization.
- **Extend to N-Dimensional Inputs:** define a taxonomy of possible input dimensions (spatial, temporal, depth, salience, geolocation, attention, provenance, etc.) and discuss how each can be co-registered with tokens to provide grounding. We examine each dimension's impact on **mutual**

information (relevance of input to task) and how it can help curb hallucination by providing *anchored context*.

- **Formal Theory – Information Bottleneck & Rate–Distortion:** lay out the theoretical guidance for designing token bottlenecks. Using the Information Bottleneck (IB) principle and rate–distortion theory [10] [11], we frame the compression of inputs as an optimization that preserves essential information for the task while discarding extraneous data. This provides a principled way to allocate a *token budget* across dimensions and content segments.
- **Architecture Design – Variable-Rate Token Bottlenecks:** propose architectures that can adaptively compress inputs. This includes a **bottleneck module** that produces a compact intermediate representation (e.g. a limited set of latent tokens) with *variable rate*—allocating more tokens to information-rich parts and fewer to redundant parts [12]. We detail mechanisms like learned token selection, multi-scale encoders, and Mixture-of-Experts gating to achieve this adaptivity [13] [14]. The design supports N-D inputs by synchronizing compression across multiple fields (e.g. compressing an image and its metadata jointly).
- **Architecture Design – N-D Synchronized Tensor Inputs:** describe how to extend transformer encoders to accept inputs with arbitrary additional dimensions. Each token can be enriched with features such as 2-D coordinates [15], timestamps, sensor readings, or other context, via specialized embedding layers or encoder modules. We introduce a **field registry** concept to flexibly plug in new input fields and ensure the model remains **synchronized across dimensions** (e.g. aligning text and layout coordinates, or video frames and time).
- **Implementation Specifications (Codex Integration):** provide an implementation blueprint for integrating these ideas into an LLM stack (code-named "Codex"). We identify the required modules (field-specific encoders, token router, compression bottleneck, etc.), detail the training objectives (combining standard LM loss with mutual-information-driven losses), and describe how to route tokens through the model (e.g. special token types or attention masks to handle different fields). The specification will enable engineering teams to build and experiment with N-D LLMs in practice.
- **Benchmarking & Experiments:** outline an empirical evaluation plan to validate the approach. This includes constructing **token-equalized comparisons** between 1-D, 2-D, 3-D, and N-D models – for fairness, each model is given a similar token budget and we measure how well each uses it. We propose benchmarks covering document understanding (layout-aware QA, table extraction), vision-language tasks (image+text QA), spatio-temporal reasoning (e.g. video understanding or timeline summarization), and structured data tasks (e.g. combining text with databases or graphs). We also discuss measuring **mutual information proxies** (like how well a model can reconstruct input or identify relevant parts [16] [17]) and evaluating hallucination rate on grounded vs. ungrounded queries.
- **Systems Integration (Memory & Orchestration):** describe how these N-D models fit into larger AI systems via a **Semantic Tensor Memory** and an **Auto-IB Orchestrator**. The Semantic Tensor Memory will log and index the model's experiences in a structured tensor format (capturing multi-dimensional context of each interaction) to enable long-term memory, auditability, and retrieval. The Auto-IB Orchestrator will monitor incoming tasks and automatically predict how to compress or route inputs (e.g. deciding which fields to prioritize) and can generate synthetic training data to continually refine the compression strategies.
- **Ethical and Safety Considerations:** analyze the implications of multi-dimensional inputs on privacy, bias, and reliability. Additional input fields (like user location or attention traces) could pose privacy risks if misused, and adaptive token selection might introduce biases (if certain information is consistently dropped). We discuss mitigation strategies – e.g. privacy filters for sensitive fields, bias audits for the routing mechanism, and safeguards to prevent new failure modes introduced by multi-modal fusion.

Through this combination of theory, architecture, empirical plans, and system integration, we aim to provide both a **scientific contribution** (outlining a new direction for information-efficient, grounded LLMs) and a **practical roadmap** for engineering teams. The following sections delve into each aspect in detail, charting a path toward LLMs that make optimal use of each token and each modality on the journey to more general intelligence.

Background: From 1-D Text to 2-D Layout – Limitations and Innovations

1-D LLM Architectures: Traditional LLMs like GPT-3 and its successors treat input as a sequence of tokens with no additional structure. They rely on positional encodings to maintain word order, but otherwise the model has no notion of spatial or temporal context beyond linear sequence. This simplicity has enabled scaling, but it also yields inefficiencies. All information must be encoded in a single sequence, often requiring repetition of context and inability to mark what is important vs. auxiliary. As context length increases, two issues emerge: **computational cost** (self-attention is $O(n^2)$ in sequence length) and **context dilution** (models pay the same attention to irrelevant tokens as to crucial ones, unless they implicitly learn to ignore). Empirically, LLM performance can degrade on very long inputs due to this dilution ("attention focal loss"), sometimes referred to as the *"lost in the middle"* effect [1] [2]. Researchers have also observed that LLMs prioritize the beginnings and ends of sequences, potentially overlooking middle content in long passages. In practice, product developers have turned to retrieval-augmented generation (RAG) to handle long contexts [4] – retrieving a few relevant chunks for the prompt – but this only addresses symptoms (by shortening inputs), not the root architecture limitations.

2-D Extensions – Layout and Vision-Augmented LLMs: To overcome the constraints of 1-D text, recent models incorporate **2-dimensional spatial structure** for document understanding. A prominent example is Microsoft's LayoutLM family, which augments text tokens with their (x,y) coordinates on the page and even visual pixel information [15] [18]. By giving the model access to layout, these 2-D models understand forms, tables, and complex documents far better than text-alone models. For instance, LayoutLM knows that a bold number at the bottom-right of an invoice is likely the total amount, because it has learned spatial patterns [19]. Empirically, including layout and visual cues yielded huge gains on document tasks – e.g. LayoutLMv2 improved key information extraction accuracy by leveraging spatial embeddings and images, and more recent layout-aware LLMs continue to push that further [20]. In essence, **adding a second dimension of position gives the model a new signal**, reducing ambiguity that would exist in a flattened sequence. This reduces errors like mixing up columns or reading order, and thus mitigates some hallucinations (e.g. the model is less likely to hallucinate a relationship between distant pieces of text if it "knows" they are in separate regions of a page).

Another line of 2-D innovation uses **vision models to encode text images**. DeepSeek-OCR (2025) exemplifies this by treating an entire document page as an image fed to a vision Transformer encoder, whose output tokens are then decoded by a language model [21]. This approach essentially moves the heavy lifting to a **2-D vision space**: since images can pack dense text in a compact form, the context length problem is alleviated. DeepSeek-OCR's encoder combines a local vision Transformer (SAM) and a global CLIP-based encoder, with a convolutional downsampler to reduce 4096 image patches to 256 tokens [22]. The decoder is a text generative model that outputs the reconstructed text. The result is a system that achieves state-of-the-art OCR accuracy while using an order of magnitude fewer tokens in the LLM's context [6] [23]. In practical terms, DeepSeek can process an entire page (complex layouts, multilingual text, even

math or diagrams) with only ~100–200 tokens of vision input, whereas a traditional OCR+LLM pipeline might require thousands of text tokens [23]. This *optical compression* showcases the benefit of **offloading context to another modality**: the LLM is no longer struggling with extremely long sequences, and it effectively delegates part of the understanding to the visual domain.

Critique of 2-D Approaches: While 2-D LLMs like LayoutLM and DeepSeek-OCR represent a major step forward, they also highlight challenges that inform our N-D design. LayoutLM's coordinate embeddings give spatial awareness, but early implementations naively concatenated coordinates as additional tokens or features, which could *increase sequence length substantially* [24]. For example, one method represents each token as "WORD [x1,y1,x2,y2]" in the input sequence [25] – this quadruples or quintuples the token count for every word (each coordinate might be a token or a few tokens). Such inflation undermines efficiency. Newer approaches like **LayTextLLM** address this by projecting each bounding box into a single embedding vector and interleaving those with text tokens [26] [27], essentially compressing the layout information to one token per word. This idea resonates with our goal of a *token-efficient representation*: one bounding-box token carries as much info as four coordinate tokens did before, reducing overhead.

DeepSeek-OCR, on the other hand, introduces a specialized two-stage architecture (vision encoder + MoE text decoder) that is highly optimized for documents [21]. It achieves remarkable compression and accuracy, but at the cost of complexity – it requires a large amount of paired image-text data and a custom training pipeline mixing OCR data, rendered synthetic data (charts, formulas, etc.), and even text-only data to retain general language ability [28]. This indicates that when moving to 2-D or multi-modal, we must carefully **balance specialization and generality**. A model like DeepSeek excels at reading images of text, but it essentially trains the LLM decoder from scratch for that purpose; one must ensure that in adding dimensions we do not lose the general language competence or other abilities of the base LLM.

In summary, the evolution from 1-D to 2-D LLMs demonstrates that: (1) Giving the model additional **structured fields** (like coordinates or pixels) can dramatically improve its efficiency and accuracy on tasks that benefit from those fields, and (2) There is a need for smart ways to incorporate these fields without exploding token counts or requiring exorbitant training data. These lessons directly motivate our exploration of **N-dimensional LLM architectures**, which generalize the notion of adding useful signals and compressing representations to achieve even greater information efficiency.

Formal Foundations: Information Bottleneck and Rate–Distortion Theory

To rigorously guide the design of token-efficient, grounded models, we turn to two key theoretical frameworks: the **Information Bottleneck (IB)** principle and **Rate–Distortion theory**. These provide a vocabulary to talk about the trade-offs between how much information from the input a model retains and how well it performs its task.

Information Bottleneck Principle: IB, introduced by Tishby et al. (1999), formalizes the idea of compressing an input X into a representation Z that preserves only the information relevant to an output target Y. In our context, X could be a complex multi-field input (text, images, etc.) and Y could be the task output (e.g. an answer or summary). The IB objective is to minimize the mutual information $I(X;Z)$ (compress X aggressively) while maximizing $I(Z;Y)$ (keep all predictive information for Y). This is typically expressed as a Lagrangian: minimize $I(X;Z) - \beta\, I(Z;Y)$, where β controls the trade-off.

Applying IB to LLM inputs means we seek a **minimal sufficient representation** of the prompt for the task at hand [10]. Intuitively, we want to throw away redundant or irrelevant parts of the input (to save token budget and reduce noise) but keep anything that could change the answer. If β is high, we prioritize fidelity to the task (risking more tokens); if β is low, we prioritize compression (risking some accuracy loss). Recent research has started using IB in prompt/context compression: Zhou et al. (2023) used IB to filter out "context noise" and focus on query-relevant content [10], and an **IB-based prompt optimizer (IBProtector)** was proposed to selectively perturb prompts to hide irrelevant info and protect the LLM from adversarial inputs [10]. These works validate that the IB view is fruitful – **it provides a principled metric of "relevance" via mutual information**. However, directly computing mutual information in LLMs is difficult; practical methods use proxies like attention weights or learned importance scores to approximate what information in X is critical for predicting Y [29] [10]. We will leverage this principle by designing training objectives that encourage the model to identify and bottleneck relevant information (for instance, using a small auxiliary model or the LLM's own cross-attention to estimate token importances [29] [10]).

Rate–Distortion Theory: While IB deals with information in a probabilistic sense, rate–distortion theory from classical compression gives us a handle on the *quantitative* trade-off between compression rate and accuracy. In our setting, the "rate" can be thought of as the number of tokens (or bits) used to encode the input, and "distortion" corresponds to the loss in performance or fidelity due to compression (e.g. a drop in answer accuracy, or an inability to recover the original input). We can ask: what is the minimal number of tokens R required so that the model's output is within an acceptable distortion (error) D from the ideal output? Researchers have started to formalize prompt compression in these terms [11]. Nagle et al. (2024) derive a **distortion-rate function** for prompts and treat prompt compression as an optimization problem: essentially, they compute the theoretically optimal way to compress a prompt for a given allowed distortion [11]. One key finding is that *query-aware, variable-rate* compression is far superior to one-size-fits-all strategies [12]. In other words, the number of tokens and which parts to compress should depend on the specific question or task – a dynamic approach. They showed a large gap between existing heuristic methods and the theoretical optimum, and introduced an adaptive method (Adaptive QuerySelect) that adjusts the compression level per query to close this gap [30]. This directly informs our design: we should allow **variable-rate bottlenecks** that can tighten or loosen depending on the input's needs, rather than a fixed compression for all cases. Rate–distortion theory also reassures us that aggressive compression is possible: many prompts have a lot of redundancy, and with an optimal strategy, one can remove a substantial fraction of tokens without changing the answer, as evidenced by 25% compression yielding the same or even better accuracy in some QA tasks [31] [32]. We can also leverage analytical tools (like the dual of the linear program in Nagle et al.) to *compute ideal token allocations* for small scenarios, which can guide heuristics or training signals in larger ones [11].

Connecting IB and Rate–Distortion to LLM Architecture: In essence, both theories tell us that there exists an optimal "bottleneck" that retains just the needed information. The challenge is how to implement that in a large model. One approach is **learning to prune or select tokens**. For example, the QUITO-X method used a small T5 encoder-decoder to generate cross-attention scores as a proxy for mutual information, then dropped low-score tokens from the context [29] [10]. Another approach is **periodic compression inside the model**: recent work on *Bottlenecked Transformers* applied IB theory to justify consolidating the Transformer's own hidden state (KV cache) occasionally [33]. They argued that a vanilla decoder-only Transformer is suboptimal because it carries forward all past token information indiscriminately, whereas an IB-optimal model would compress intermediate results to focus on what's relevant for future predictions [34] [35]. By adding a **Cache Processor** that rewrites and compresses the hidden state at certain boundaries (analogous to how the brain consolidates memory), they achieved better

reasoning performance with fewer tokens [36] [37]. This insight is transferable to input processing: instead of feeding an LLM an entire long input at once, we can imagine **segmenting and compressing** it in stages (hierarchical encoding). In fact, human reading often works this way – we summarize or highlight as we go through a document.

In conclusion, IB and rate–distortion theory give us a **north star**: we want our LLM to encode the *minimal sufficient statistics* of the input for the task. This means implementing a learned bottleneck that throws away noise and redundancy. The next section translates this into concrete architecture components that can achieve variable-rate compression and handle multiple input modalities, all while maintaining high task performance (low distortion).

Architecture Design: Variable-Rate Token Bottlenecks

A core proposal of our framework is to introduce an **adaptive token bottleneck** in the LLM's input processing pipeline. This bottleneck constrains the amount of information (number of tokens) passed to the later stages of the model, forcing an explicit compression guided by task relevance. Crucially, it is *variable-rate*: the model can use more tokens when the input is complex or the task demands high fidelity, and fewer tokens when parts of the input are less informative or redundant [12]. This dynamic allocation is key to achieving near-optimal usage of the model's capacity for each query. We now describe the design of this bottleneck mechanism.

Two-Stage Encoder with Compression: We split the LLM's input processing into a **High-dimensional Encoder** (stage 1) and a **Bottleneck Transformer** (stage 2). In stage 1, the model ingests the raw multi-field input (text, images, coordinates, etc.) and produces a rich representation. This could be implemented with specialized sub-modules per modality: for example, a CNN or ViT for images, a small language model for text segments, or a graph neural net for structured data. At the end of stage 1, instead of concatenating all features into one long sequence, we insert a **compression module**. This module takes the full set of stage-1 outputs (which might be thousands of token embeddings) and outputs a much smaller set of *summarized tokens*. Conceptually, it performs a learned "summary" or "selection" of the input. Several strategies can realize this:

- *Learned Summarization Tokens:* Provide the model with a fixed number of trainable summary tokens (like learnable queries) that attend over the stage-1 outputs and distill information. For instance, we could have 128 summary tokens that each attend to different parts of the input and capture essential info. This is similar to how Transformers sometimes use a CLS token for classification, but here we'd use multiple and train them to cover the input space. During training, a loss would ensure the final output can be predicted from these summary tokens (with perhaps an auxiliary decoder trying to reconstruct some input details to enforce coverage). This approach yields a fixed bottleneck size (e.g. always 128 tokens), which can be efficient but might waste capacity on simple inputs or be insufficient for very complex ones.

- *Dynamic Token Selection (Variable Number):* To achieve a truly variable rate, we allow the number of tokens after compression to vary. One implementation is to use an attention-based scoring to rank the importance of each stage-1 output token for the task, then keep the top-k tokens. The value of k could be chosen by a small network or heuristic that looks at the distribution of scores (for example, choose k such that the cumulative importance is 99%). Alternatively, we can use a threshold on importance – all tokens with importance above a certain threshold pass through. This

importance scoring could come from a **cross-attention with the query** (if the task provides a distinct query, like in QA) as demonstrated in QUITO [29] [10], or from the model's prediction loss gradients (as a measure of each token's influence on the output). The challenge is to make this differentiable for end-to-end training. Techniques from sparse attention and differentiable top-k selection (e.g. using softmax with a temperature, or perturbation methods) could be applied. The benefit is that the model can use, say, 50 tokens for one input and 300 for another, adjusting to content. This aligns with the rate–distortion notion of using just as many bits (tokens) as needed for a given distortion level.

- *Hierarchical Compression:* Combine the above by compressing in multiple steps. For instance, a long document could be broken into chunks, each chunk compressed to a few tokens, then those tokens further compressed. This mimics human note-taking and could be easier to train (local compression followed by global). A hierarchical approach also meshes well with **multi-resolution encoders**; indeed, DeepSeek-OCR already used multi-scale vision encoding (16× patch compression then global attention) [38]. We can generalize this: e.g., first reduce an image to regional tokens, then have another stage that selects only salient regions to form final tokens for the decoder. Each stage acts as a bottleneck, potentially with its own IB loss (ensuring it doesn't drop needed info).

Prompt-Conditioned Mixture-of-Experts (MoE) Routing: One powerful architecture pattern for variable computation is MoE, where multiple expert networks are available but a router activates only a subset per input. We propose to incorporate MoE at two levels: (1) **Modality-level experts** and (2) **Token-group experts**. Recent work on multiparametric MRI VQA introduced a prompt-conditioned MoE that decides which imaging modalities are needed for a question [13]. They had experts for each MRI sequence and a router that activates only relevant ones per query. Inspired by that, our architecture can have modality experts (text expert, vision expert, etc.) and a router (in stage 1) that allocates more capacity to the modalities that matter for the task at hand. For example, for a purely textual question, the image expert might be mostly bypassed, saving computation and not flooding the bottleneck with image tokens. Conversely, a visual question would engage the vision expert strongly. This is **adaptive fusion of modalities** – unlike static early fusion of all inputs, it prevents irrelevant fields from consuming token budget. At a finer granularity, within a modality's outputs, we can use MoE experts to compress tokens. The mpLLM system had both modality-level and token-level experts for merging multi-3D images into one representation [39] [13]. In our design, we could have, say, multiple "compression experts" each specializing in a certain type of content (e.g. one expert tuned for spatial data, one for temporal sequences, one for tabular structures). The router (which could be another small neural network looking at an overview of the input) would assign different parts of the input to different experts for compression. The outputs of these experts then form the bottleneck tokens. Such a weighted combination, *conditioned on the prompt*, has been shown to outperform single static compression in ablation studies [14] [40]. It gives the model flexibility to apply different compression strategies depending on whether it's, say, parsing a diagram versus summarizing a story.

Maintaining a Continuous Information Measure: To train the compression module, we need a signal for how much information is retained. We can approximate $I(Z;Y)$ by the model's loss on the final task (since if Z loses info, loss will increase) and $I(X;Z)$ by something like the average entropy of Z or the number of tokens passed (as a proxy for bits). One practical approach is to add a *penalty term* proportional to the number of tokens after compression (to encourage fewer tokens) and adjust its weight to achieve the desired trade-off. Alternatively, use a variational IB: impose a prior on Z (like Z should be sampled from a simple distribution) and add a KL divergence term between the encoder output distribution and the prior.

This is done in VIB (Variational Information Bottleneck) approaches. For example, we can treat the compression as stochastic: each stage-1 token is kept with some probability p (learned per token), then $I(X;Z)$ relates to the entropy of that binary mask. Training would then encourage making many p either 0 or 1 (drop or keep) and penalize keeping too many with an information cost. While the implementation details are complex, the high-level effect is that the model *learns to drop tokens that don't affect the final loss much*. Over time, it might learn to drop boilerplate text, repetitive sentences, or irrelevant metadata, while keeping tokens that change the answer or output.

Integration with Decoder/LLM: Once the bottleneck module outputs its compressed token set, these tokens are fed into the main LLM decoder (stage 2). The decoder then produces the final answer or continuation. Since the decoder is now getting a much shorter input, its attention computation is manageable even for what originally were very long inputs. One might worry that compressing will lose information – indeed, if done poorly, the model could drop something important and be forced to guess (hallucinate). But by training jointly, the compression module can be optimized to preserve what the decoder needs. In fact, we can have a **dual loss**: one for the final task output and one for input reconstruction. For instance, during training we could occasionally ask the model to output not just the answer but also reconstruct some part of the input (like key facts) from the compressed tokens, ensuring the bottleneck hasn't lost them. DeepSeek-OCR did something similar by explicitly training the decoder to output the full text from the image tokens [41]. In our case, we might not want full reconstruction (especially if input is huge), but selective reconstruction or consistency checks (e.g. if the question is about a date in a document, ensure that date token is preserved through the bottleneck by checking if the model can output it when prompted). These are like **mutual information proxies** – if the model can regurgitate certain details on demand, those details must have passed through Z.

In summary, the variable-rate token bottleneck architecture introduces a smart filter between input and model. It can be thought of as a **learned encoder that right-sizes the context** for the LLM. This encoder uses mechanisms like summarization tokens, dynamic selection, and MoE routing to adaptively compress the input, guided by IB/rate-distortion objectives. The next part will expand how this fits with **N-dimensional inputs**, because designing the bottleneck goes hand in hand with leveraging the structure in multi-field data.

Architecture Design: N-Dimensional Synchronized Inputs

Extending LLMs to N-dimensional inputs means allowing the model to consume data that has additional structure beyond a single sequence. Here *N-dimensional* broadly refers to any set of **co-registered fields or modalities** that come with the input. For example, a **3-D input** might be an image (with 2-D spatial coordinates) plus time (making it video, a 3rd dimension), or text with an associated timeline of events. More generally, we consider inputs like: text tokens with spatial coordinates (2-D layout), video frames with time (3-D: two spatial + time), images with depth maps (3-D: x,y,depth), sensor readings with geolocation and timestamp (maybe 3 or 4 fields: value + lat + long + time), or even each token having an "importance" score from an external model (an additional dimension of salience). The **taxonomy of dimensions** can be seen as:

- **Spatial (2-D position):** e.g. document layout coordinates for text, pixel locations in an image.
  Impact: Preserves layout and grouping information. High mutual information for tasks where spatial arrangement conveys meaning (forms, tables, UI understanding). Reduces hallucinations such as conflating content from different sections, since model knows spatial boundaries [9] [15].

- **Temporal (time sequences):** e.g. timestamps in transcripts, chronological ordering of events, video frames. Impact: Conveys sequence beyond static text order (e.g. real time intervals). Helps maintain causal or temporal consistency; model is less likely to hallucinate an incorrect timeline or causal relation if time is explicit. Also allows queries like "before/after X" to be answered from data.
- **Depth/Geometry (3-D space):** e.g. depth sensor input for each pixel, 3D coordinates of objects, or any spatial geometry data. Impact: Adds physical context – in vision, depth can help disambiguate object sizes or occlusions. For LLMs describing scenes, having depth info could prevent hallucinating incorrect spatial relations (like assuming something is visible or larger than it is). In text domains, "geometry" could mean structural information (like parse trees or program AST nodes), giving another dimension of context.
- **Modality/Channel:** if an input has multiple modalities or channels (like MRI with multiple imaging sequences, or a multi-sensor fusion scenario), we treat each as a dimension. E.g. in mpMRI VQA, they had T1, T2, FLAIR MRI sequences – each is a 3D volume but co-registered spatially. An N-D model can incorporate a "modality index" to distinguish them but also align them. Impact: The model can learn inter-dependencies between modalities (e.g. certain features only show up in one channel). It reduces hallucination in answers by cross-verifying – if one modality doesn't confirm a finding, the model might be less confident. This was evidenced by mpLLM's ability to leverage inter-modal information and its high out-of-scope question detection accuracy, meaning it didn't hallucinate answers when info wasn't present [7] [42].
- **Salience/Attention Maps:** an input field that flags which parts of the input might be important. This could come from an external model (like a cheap model highlighting likely relevant bits, or user highlighting in an interface). It could be as simple as a binary mask per token or a continuous weight. Impact: Acts as a prior for the bottleneck – the model can choose to trust or at least consider the salience hints when compressing. It can speed up convergence of the compression learning (similar to how human annotations can guide attention). However, there's a risk: if the salience model is biased or wrong, it might cause the LLM to drop truly important info and amplify biases. We must allow the LLM to override or ignore salience signals when needed. Still, as an added dimension, it explicitly provides a notion of "importance" that can align with mutual information objectives [43] (since less informative tokens can be marked as low salience).
- **Provenance/Source Metadata:** information about where each token/piece of data came from – e.g. document ID, author, reliability score, sensor ID, etc. In retrieval-augmented contexts, one might tag each retrieved sentence with the source. Impact: This can help the model assess credibility and resolve conflicts. For example, if a piece of text comes with a tag "Source: Wikipedia" vs "Source: random blog", a well-trained model could down-weight the less reliable source when synthesizing answers, thus reducing hallucination or incorrect assertions. Additionally, provenance tags enable the model to output citations or at least trace its answers, which is important for trust. As a dimension, provenance might be represented by learnable embeddings corresponding to source IDs or categories (like a vector for "academic journal" vs "forum post"). The model could learn a kind of attention bias: trust but verify certain sources. Ethically, this field is sensitive (we don't want source info to introduce unfair bias), but it's crucial for groundedness.
- **Modality-specific coordinates:** e.g. frequency in audio (for spectrograms), latitude/longitude in maps. These can be considered additional spatial dimensions specific to certain data. For instance, a **geo-aware LLM** might take input: "Text of a news article" + the latitude/longitude of where it was published, or a range of a map it's about [44] [45]. With coordinates, it could answer spatial questions or incorporate knowledge of that region's context. Likewise, if analyzing audio transcripts, knowing the time-frequency info of speech could help separate speakers (like each word could carry a speaker ID from an audio model). Essentially, any structured attribute can be an input field.

Designing the Model to Ingest N-D Inputs: At the fundamental level, a Transformer can ingest a sequence of tokens, each a vector. To make it N-D aware, we integrate the extra dimensions into those vectors. A general recipe is: **Embed each dimension and combine**. For each token (or element of input), we form an embedding as the sum (or concatenation + linear projection) of: - Content embedding (e.g. word embedding or image patch embedding). - Position embedding for each dimension (e.g. 1-D position in text, plus 2-D position if applicable, plus maybe a time embedding if token has a timestamp, etc.). - Field type embedding if different fields produce separate tokens (e.g. a token might be marked as coming from the "title" field vs "body" field of a document, or "vision" vs "text" domain).

For 2-D, LayoutLM did exactly this: a word token's final embedding = word vector + x-pos embedding + y-pos embedding, plus a segment embedding distinguishing if it's from text or an image overlay [15]. We generalize this: for a token with coordinates (x,y,z) and time t, we add embeddings for each. If a token lacks a dimension (e.g. a pure text token has no (x,y)), we can just add zeros or a default embedding for that dimension. A field like salience (a scalar importance) could be discretized into a few levels and embedded or directly used to scale the token's content vector (like an attention bias). In practice, these embeddings need to be learned, and careful normalization is required to ensure no single dimension dominates.

Synchronized Multi-Modal Encoding: Sometimes different modalities come in different token streams (e.g. an image yields a grid of patch tokens, text yields a sequence of word tokens). A naive approach is to just concatenate all tokens and add type embeddings. However, this might lose the alignment between modalities. For instance, in a document, an image of a chart and the caption text are related – their tokens should attend more to each other. Synchronized encoding means preserving some structure in the attention pattern or token ordering that reflects the N-D structure. One approach is **interleaving**: e.g. always follow a block of text tokens with its corresponding layout coordinate token (as done in coordinate-as-token schemes) [25]. Another is to use **factorized attention**: e.g. allow the model to attend more freely among tokens that share the same spatial coordinates or time stamp. There has been research on disentangled attention for layout (treating layout as a separate sequence to attend on) [46]. We can include *relative bias* terms: if two tokens are close in 2-D space, add an attention bias encouraging the model to connect them. Similarly, tokens with the same timestamp or same source might get a bias. These details would be part of the field registry configuration – e.g. a field can specify if it should induce any special attention patterns.

Example – Document with Layout and Salience: To make it concrete, imagine an LLM input consisting of a document with text and layout, plus an external model's salience score for each text box. For each word token, we create: word embedding + (x,y) embedding + salience embedding. The salience could be a small set of embeddings like {not highlighted, highlighted} or a continuous value embedded via a small MLP to a vector. The transformer sees a sequence of such enriched tokens. Now, suppose a section of the document is irrelevant to the query – ideally, the salience model marked it low, and the learned positional embeddings also encode that it's far from where relevant info (say, a specific table) is on the page. The transformer's self-attention might learn to focus on tokens with certain salience or within certain coordinate ranges, essentially performing a first-level filtering. This, combined with the later bottleneck compression, means that those irrelevant tokens are likely dropped before final decoding. On the other hand, if some important text is small or out of order in raw text but positioned prominently on page, the spatial embedding helps the model realize its significance (maybe by correlating historically that important info often appears at top or in bold regions). This enriched understanding is hard to quantify but often shows up as improved accuracy on layout-sensitive tasks [9].

Taxonomy Impact on Mutual Information and Hallucination: Each dimension added can increase mutual information $I(X;Y)$ if that dimension carries signal for the task. For example, adding time stamps (X gains time) will increase $I(X;Y)$ for any question about temporal ordering. If previously the model had to guess the order of events (prone to hallucination), now it's grounded by time. Similarly, adding source provenance increases $I(X;Y)$ for questions of trust ("was this statement from a reliable source?") – previously the model might hallucinate an answer or be unsure, now it has explicit data. More information usually helps, *but only up to a point*: if a dimension is noisy or irrelevant, it can *reduce* effective mutual info by adding noise. For instance, adding a poorly-made salience map that sometimes highlights wrong text could confuse the model (decreasing $I(Z;Y)$ after compression because the model latched onto a wrong highlight and threw away right info). Thus, each field should be used with care.

From a hallucination perspective, hallucination often occurs when the model's input under-specifies the output – the model then fills in from training priors. N-D inputs aim to *over-specify* the situation by giving multiple corroborating signals. A multi-modal LLM that sees an image and text will be less likely to hallucinate a detail about the image because the image acts as ground truth (unless the image is ambiguous). In experiments, multi-modal models have shown fewer free-form fabrications when the image clearly shows something – instead, they might err by misidentifying, which is a different failure (perception error rather than hallucination). In the mpLLM medical example, the model was trained to abstain if it didn't have evidence in the MRI data [7]. Having multiple modalities also lets the model check consistency; it might be reluctant to output something unless both text and image modalities agree on it. The taxonomy also suggests new ways to catch hallucinations: if the model has a provenance field saying "this fact is from source X", it can be constrained to not output facts that aren't linked to any source in the input (like a built-in citation requirement, similar to how retrieval-augmented models avoid hallucination by forcing evidence).

To conclude this section, our N-D architecture treats every input as a **tensor of features** rather than a plain sequence. Through positional and field embeddings, careful attention design, and possibly field-specific encoders, we ensure the model can make sense of each dimension. All fields are synchronized in the sense that they refer to the same underlying content (co-registered), and the model learns to **integrate them to enrich understanding**. This multi-dimensional encoding feeds into the variable-rate bottleneck described earlier: importantly, the bottleneck can leverage the extra dimensions to compress smarter. For example, it might drop tokens that are marked with low salience or from sources deemed irrelevant, or compress image regions that are empty background while preserving regions containing text – decisions it could not make without those signals. Thus the N-D input and the bottleneck compression work in tandem as a pipeline for maximal efficiency.

Implementation Specifications for the Codex Prototype

Translating the above concepts into a working system requires defining concrete modules, data flows, and training procedures. Here we outline an **implementation blueprint**, which we'll refer to as the *Codex* prototype (not to be confused with OpenAI's Codex; here it's just a project code name). The goal is to provide enough detail that an engineering team could start building the system and iteratively refine it.

Modular Architecture: The system is organized into modular components, each responsible for a certain field or function. Key modules include:

- **Field Encoders:** Each input field type (text, image, etc.) has a dedicated encoder that transforms raw data into token embeddings. For text, this could be a tokenizer + embedding matrix (possibly

initialized from a pre-trained LLM for language). For images, it might be a Vision Transformer (ViT) or convolutional network that produces patch embeddings [38]. For other modalities like audio or time series, appropriate encoders (e.g. a spectrogram CNN or an RNN) are used. Each encoder outputs a sequence of vectors. Crucially, along with each vector, we carry field identifiers and coordinate metadata. The encoders should also output these, or we infer them (e.g. the image encoder knows the (x,y) location of each patch it outputs; a text encoder might know page number or section).

- **Field Registry:** This is essentially a configuration that lists all fields, their data types, and how to embed their coordinates. For example, the registry might say: field "text" – uses tokenizer X, add 1-D positional embedding; field "layout_x" – numeric, use learned 128-dim embedding; field "layout_y" – numeric, 128-dim embedding; field "image_patch" – uses ViT backbone Y, add 2-D position embeddings; field "source_id" – categorical, use 64-dim lookup per ID. The registry allows easy addition of a new field; one just provides an encoder and defines how to embed its positional attributes. The model's code then automatically includes those in the input embedding composition. This design ensures **extensibility** – tomorrow if we want to add a "depth" channel, we implement an encoder for depth maps and update the registry.

- **Token Merger:** If multiple fields produce tokens, we need to merge them into a single sequence for the Transformer. The merger takes the outputs of all field encoders and intermixes them. There are options: concatenate in a fixed order (all text tokens, then all image tokens, etc.), or interleave by some key (e.g. by spatial proximity or time). A simple approach is to concatenate but include within each token's embedding a field type indicator (so the model knows, e.g., token 5 is an image token). We also insert special separator tokens if needed to mark boundaries (like a <image> token at start of image tokens – DeepSeek uses `<image>\n` tags in its prompt format to mark where image data goes [47]). The exact scheme might be adjusted based on empirical performance. Importantly, the merging must be **deterministic given the same input** (so that we can consistently align, especially for tasks where order matters). The token merger might also add initial special tokens like a global CLS or a task descriptor token if needed.

- **Compression Bottleneck Module:** As designed earlier, this module receives the merged token sequence (with all embeddings combined) and outputs a compressed sequence. In implementation terms, this could itself be a Transformer or a sequence of transformers. For instance, one might use a **Perceiver IO** style encoder: cross-attend a fixed small set of latent bottleneck vectors to the input tokens. Alternatively, implement the learned summarization tokens approach: initialize m trainable summary tokens that attend to the input (via one transformer block), then those summary tokens (size m) become the compressed representation Z. We could also use multiple layers of this attention to refine the summary. For variable m, we might set an upper bound M and allow an attention mask to effectively drop some (like summary tokens that just don't get used could be masked out). Another approach is iterative clustering: the module could perform something like pooling – e.g. divide tokens into k groups and average each group (with learned weights) to form k tokens. This is like a differentiable clustering or low-rank projection. The architecture choice will be guided by experiments; a safe starting point is using a transformer layer or two that maps N input tokens to M output tokens ($M < N$). Many libraries support transformer decoder layers that can do cross-attention from a smaller set of queries onto the input keys/values – that's a direct way to implement a learned summary.

- **LLM Decoder (Backbone):** After compression, we have a set of Z tokens. These are fed into a standard LLM decoder (which could be pre-trained GPT-like or a smaller custom model). If using a pre-trained LLM, one has to adapt it to accept our token types. One strategy is to project our compressed tokens into the same dimension as the LLM's embeddings and then either prepend or insert them into a prompt for the LLM. For example, we could format it as: `<compressed> ... tokens ... </compressed> Query: ...`. However, it's cleaner to integrate at the model level: essentially treat the compressed tokens as the "context" for the decoder. If we have the source of the LLM (like an open-source model), we can modify its architecture to have an encoder-decoder style: our compression module is like an encoder, and the LLM's layers attend to those encoder outputs (like how T5 or other seq-to-seq models work). In fact, designing the system as an encoder-decoder model might be ideal for training from scratch or fine-tuning: the encoder handles multi-dimensional input and compression, the decoder generates text. This way, we can leverage techniques from sequence-to-sequence training and even initialize the decoder with a pretrained language model (converted to a decoder-only that cross-attends to encoder outputs). Recent multi-modal LLM frameworks (like PaLM-E or Flamingo) often use a similar approach where a pretrained LM is augmented with cross-attention to image features. We will adopt that pattern for integration: the compressed tokens Z will serve as keys/values for the decoder's cross-attention layers.

- **Loss Functions:** We optimize the model on multiple objectives:

- **Primary Task Loss:** If the model's purpose is QA or text generation given the input, the main loss is the negative log-likelihood of the correct output text (or a suitable reward if using RLHF or similar for open-ended tasks). This ensures the model learns to produce the right answers from the compressed representation.
- **Compression Loss/Penalty:** We add a term to encourage smaller Z. One simple formulation is $L_{\text{compress}} = \alpha \cdot \frac{|Z|}{|X|}$ or just $\alpha |Z|$, i.e. penalize the number of tokens in Z. Another is to penalize the information content: $\alpha I(X;Z)$. In practice, $I(X;Z)$ can be upper-bounded by $\sum_{z \in Z} H(z)$ (the entropy of each compressed token) assuming some independence. We could thus impose a penalty on the entropy of each bottleneck token distribution. If we make the compression module probabilistic (like it outputs a mean and variance for each token embedding and we sample), then a KL divergence to a prior (like a standard normal) would serve as an info bottleneck penalty – this is analogous to a Variational Autoencoder's KL term. For initial implementation, a length penalty with scheduled weight (increasing over training) might be simpler.

- **Reconstruction/Consistency Loss:** Depending on tasks, we might incorporate an auxiliary objective to reconstruct certain aspects of the input. For example, in training, we could ask the model to output a brief summary of the input or to answer additional questions that test different parts of the input, ensuring that information isn't lost. In an extreme case, train it as an autoencoder (input to output same input) for a portion of steps to force near-lossless compression, then fine-tune for task. However, pure autoencoding could be wasteful if we truly want lossy compression focusing only on relevant info. A compromise: only reconstruct salient content. If we have labels for important pieces (like key facts or answers to potential questions), we ensure the model can output those from Z. This acts as a *mutual information proxy*: if Z retains those facts, $I(Z;Y)$ for relevant Y is high. We can leverage the salience field if available – e.g. if salience highlights something, have the model explicitly output it in a special training mode ("extract highlighted text"). This ties into the **Auto-IB Orchestrator** generating synthetic tasks (discussed later) that probe for information retention.

- **Regularization and Others:** We keep standard regularization like weight decay, dropout. If using MoE, ensure load-balancing losses (common in MoE to ensure not all inputs use the same expert) are included [14]. If any field is not always present (like sometimes images missing), add losses to handle missing gracefully (e.g. if no image, the image encoder outputs a single `[NoImage]` token – train the model to handle that).

- **Training Regime:** Likely a mix of pre-training and fine-tuning. If starting from scratch, we'd need a large multi-modal dataset (e.g. documents with text+images+QA pairs). A practical approach is to start from a pre-trained LLM (for language) and possibly a pre-trained vision model, then *jointly fine-tune* on a smaller curated dataset that exercises multi-dimensional capabilities. For example, create a training corpus that includes:

- Document QA (to use text + layout)
- Image captioning or OCR tasks (to use vision)
- Timeline summarization (to use temporal data)
- Synthetic tasks where certain info is present or absent to train the model when to say "I don't know" (to reduce hallucination)
- Perhaps use existing multi-modal datasets (like DocVQA, OCR-VQA, medical reports with images etc.).

Additionally, use the **Auto-IB Orchestrator** to generate data (discussed below) that specifically challenges the compression. For instance, take some text, generate a question, then give a long irrelevant passage plus the relevant snippet, training the model (with IB loss) to drop the irrelevant part and still answer correctly – this simulates the context filtering behavior with known ground truth.

- **Inference Pipeline:** At runtime, the system will:
- Take user input (which could be multi-part, like a query plus a document, or an image, etc.).
- Through the field registry, route each part to the correct encoder.
- Merge the token sequences with embeddings, run the compression module to get Z.
- Feed Z to the decoder and generate output (answer/solution).

- Possibly consult the Semantic Memory or external sources if needed (depending on integrations, see later section). Because of the compression, we expect inference to be faster than a baseline that concatenates everything. For example, if a document had 5k tokens and we compress to 500 tokens, the decoder part is 10× faster (since attention is on 500 tokens, not 5000). The encoder part adds some cost (e.g. vision encoding), but if using efficient encoders (or hardware offloading like CNNs on GPU), it should be manageable. There's a trade-off threshold where if the input is extremely long, the compression provides huge wins.

- **Evaluation during development:** To ensure the implementation works, we will track metrics like compression rate vs. accuracy. For various tasks, measure how many tokens the model uses relative to input length and what the performance is. Monitor the distribution of token usage per field (maybe the model learns to drop images for certain queries, etc.). We should also test extreme cases: e.g. if an input is entirely irrelevant to the query, does the model compress it down to almost nothing and correctly say "irrelevant" or produce a refusal? Conversely, if input is critical, does it keep most of it? Such behaviors can be debugged by looking at attention patterns in the compression module – e.g. check that important tokens indeed got high attention weights.

The implementation as described will likely iterate: initially, perhaps implement a simpler fixed-size bottleneck, get that working, then add variable rate. The field registry and modular encoders make it easier to add one piece at a time (start with text+layout, then bring in images, etc.). Throughout, we maintain compatibility with pre-trained weights where possible (to leverage existing knowledge) – e.g. we could use a pre-trained text encoder or at least word embeddings, and a pre-trained image backbone.

By specifying the above, we ensure the engineering team has clear targets: implement encoders, implement the bottleneck (with IB loss), integrate a decoder, then gradually increase dimensionality and adaptivity. Next, we discuss how to evaluate this system scientifically in a benchmark setting to verify the benefits.

Benchmarking and Experiment Plan

A critical component of this project is validating that N-dimensional, bottlenecked LLMs indeed offer better information efficiency and grounding than standard approaches. We propose a comprehensive benchmarking strategy with two main principles: **token-equalized comparisons** and **multifaceted evaluation** (accuracy, efficiency, hallucination, etc.).

Token-Equalized Comparison: One potential confounder when comparing models is that a multimodal model might simply use more tokens or parameters than a text-only model. To isolate the gains from better information utilization (rather than just scale), we design experiments where each model is given roughly the same "token budget" or computational budget, and we measure performance. For example, consider a long document QA task: - *Baseline 1-D model:* can take up to N=2048 text tokens (due to context window limit). - *Our N-D model:* can take the document as an image + text layout. Suppose the image+layout yields 300 tokens after compression. We will allow the baseline to also only use 300 tokens of text (which likely means it can't even include the whole doc, it might have to truncate or summarize). - *Comparison:* If our model at 300 tokens outperforms the baseline at 300 tokens, it shows the multi-dimensional compression was effective. We can also compare to baseline at full 2048 tokens (if baseline can even handle it) – often baseline might score higher with full context, but at huge cost. The idea is to demonstrate that **for a given computational cost, the N-D model retains more useful info**.

Concretely, we will run experiments varying token budgets (100, 200, 500, 1000 tokens) and plot performance. We expect curves like: the baseline's accuracy might jump only when tokens > 800 (because it needs that many to fit the needed text), whereas our model might reach similar accuracy with 200 tokens (having compressed layout effectively). This yields a rate–distortion style curve empirically. One benchmark example: the **Fox Compression Test** used by DeepSeek, where they measured OCR precision as a function of compression ratio [16]. We would replicate similar tests: e.g. take a document and ask the model to reconstruct text from increasingly compressed representations (5×, 10×, 20× compression) and see where it fails. In DeepSeek's case, 10× was nearly lossless (~97% accuracy) and 20× still had ~60% [6]. We can compare our method to theirs if possible (maybe simpler if using their open model) or at least show how adding more dimensions (like combining text+image+layout) can push that boundary further.

Benchmarks Selection: We choose tasks that cover 1D, 2D, 3D, and N-D scenarios: - **Long Text Comprehension:** e.g. *TriviaQA* or academic QA where answers are in a long wiki article. Baseline would either truncate or use retrieval. Our model could compress via IB. Metric: Exact Match or F1 of answer, vs number of tokens used. This tests general long-text handling. - **Document Understanding:** e.g. *DocVQA (Document Visual QA)*, *InfographicVQA*, or form understanding tasks. These require layout understanding. We

compare a text-only model given OCR (maybe concatenated text) vs our model given text + coordinates or an image. Metrics: accuracy on questions. We expect our model to particularly shine on spatial questions (like "what is the value in the cell above X?" where spatial reasoning is needed). - **Table comprehension:** tasks like *WikiTableQuestions* or *TableQA*, or even simple arithmetic on tables. Represent a table as text vs as a 2D grid input. See if model with 2D awareness performs better in consistency (not mixing rows). We can also measure if hallucinations (like making up a table entry) are reduced because the model sees an explicit structure. - **Vision-Language Tasks:** e.g. *VQA (Visual Question Answering)* on images, or *NLVR2* (which has images and text statements) – we can incorporate image and text. Baselines might be a vision+text model (like CLIP+GPT). Our advantage would be if we add extra fields like region captions or depth info. For instance, if we have an autonomous driving dataset with images + LiDAR (depth), we can test a model's ability to answer questions about distances or counts. A model with depth field should outperform one without when asked "how far is object A from the camera?" because otherwise it must infer from appearance (prone to hallucination). - **Spatial/Temporal Reasoning:** e.g. a *Video QA* dataset (like TGIF-QA or how/why questions about short videos). Represent video frames as images with time stamps. Baseline might sample a few frames or treat it as image+text. Our model can incorporate the time dimension to know sequence. Evaluate if it gets temporal order questions right more often. Another is *timeline summarization*: give the model a sequence of dated events and ask for a summary or cause-effect analysis. The time dimension explicitly might reduce mistakes like summarizing out of order. - **Geometry and Diagrams:** There are synthetic tasks where an input is a diagram or geometric description and questions require understanding it. For example, a puzzle with points on a plane or a simple floorplan. If we encode coordinates as input, does the model handle "nearest point" questions correctly vs having to guess or approximate by text alone? We could devise a custom evaluation where we give coordinates of objects and ask spatial questions. The N-D model can incorporate those coordinates numerically, whereas a text model sees just descriptions ("point A is at (2,3)…") which might be harder to reason precisely about. - **Multi-Modal Information Extraction:** For instance, a dataset where each sample includes text, image, and metadata and the task is to produce a structured output combining them (like filling a report from both an image and text). We can test consistency: does the model leave fields blank if info is missing (good behavior) or does it hallucinate? For example, an ID card parsing task: if we provide an image of an ID and some text fields, the model should extract name, date, etc. If some field isn't visible, an ungrounded model might guess a plausible value (hallucinate), but a grounded model should either leave it or say "unknown". We measure the rate of incorrect hallucinated values.

Mutual Information Proxies: We will measure how well the model retains information via tasks like: - **Query Relevance Ranking:** Given a long input and a query, output which parts of the input were used. If our model is compressing appropriately, it should implicitly do this. We can explicitly test by asking it to highlight relevant sentences. Alternatively, measure the overlap between the model's answer and the ground truth supporting text (did it capture the needed facts?). This can be done with metrics like ROUGE overlap with relevant parts. - **Out-of-Scope Detection:** As done in mpLLM, evaluate if the model correctly responds with uncertainty when an input doesn't contain the answer [7]. For example, ask a question unrelated to the document/image provided. A well-bottlenecked model ideally would realize nothing relevant passed through Z and thus say "I don't know" or refuse. A hallucination-prone model might just invent an answer. We can quantify this: feed some inputs where answer is absent and see percentage of times model correctly indicates inability vs produces a specific (likely wrong) answer. The 99.8% accuracy mpLLM achieved on detecting out-of-scope tasks [7] is a high bar; we'd like similarly strong performance, indicating the model isn't hallucinating from thin air. - **Compression Rate vs Performance Curve:** As mentioned, we generate curves of how performance drops as we force more compression. For instance, gradually limit the number of tokens allowed in Z and see where quality degrades. This can be compared

to baselines using summarization (e.g. GPT-4 summarizing then answering). If our method follows the theoretical rate-distortion curve more closely, it's evidence of efficiency. We might also compare to trivial compression like taking the first N tokens of text – to show our learned approach retains the most useful info rather than arbitrary truncation.

A/B Testing with Human Evaluation: In addition to quantitative benchmarks, for generative tasks (like summarization or open-ended Q&A), we can do human eval or pairwise comparison between outputs of baseline and our model. We'd look for things like factual accuracy, completeness, coherence. We expect our model's summaries to be more grounded in the source (because of multi-dimensional awareness of context) and less prone to adding extraneous details. Human judges can be asked which summary is more faithful to the document, etc. If we include provenance in the model, we could also have it generate citations for facts, and measure correctness of those citations (like how often the cited source actually supports the statement). A grounded model should cite correctly if it uses the provenance dimension well.

Benchmark Execution and Reproducibility: We will create a script for each benchmark to run the baseline vs our model under controlled settings. For fairness, ensure both models have similar parameter counts (if not, we note differences) and run enough trials for statistical significance. We also consider fine-tuning both on the specific tasks to see the achievable performance, but the main interest is zero-shot or few-shot because multi-dimensional grounding should help especially in zero-shot (where baseline might not know how to handle layout at all, whereas our model can rely on learned general layout skills). We will report results in tables and plots, for example:

- Table: **Token Budget vs Accuracy** on DocVQA, for baseline and our model.
- Plot: **Compression Ratio vs F1** on long QA.
- Table: **Hallucination Rate** (percentage of answers with unsupported content) for baseline vs our model on a set of grounded QA.
- Table: **Benchmark scores** (like exact match, or BLEU) across tasks (document QA, image QA, etc.), highlighting where adding certain fields boosted performance.

We also intend to release a *benchmark suite* specifically for N-D evaluation. This might involve constructing a new evaluation called **Token Challenge Benchmark**: for instance, present a model with a series of tasks with increasing context size and see how it scales. Or a **Multi-field QA** where some questions need text, some need image, some both – to test how well models can pick which modality to use. SpatialLLM introduced a dataset for spatial QAs [48] ; similarly, we might curate a small set of multi-field questions to test combined reasoning (e.g. a question that needs both a chart image and some text to answer).

By executing this experimental plan, we aim to empirically demonstrate: - The efficiency gains: our model achieves the same accuracy as baseline using fewer tokens (or higher accuracy with same tokens). - The improved grounding: lower hallucination rates and better out-of-scope refusal behavior. - The versatility: capability to handle tasks across different domains (documents, vision, spatial, etc.) within one architecture, indicating a step toward more general intelligence as opposed to siloed models for each modality.

System Integration: Semantic Tensor Memory and Auto-IB Orchestrator

Developing a powerful model is only half the battle – integrating it into real-world AI systems and workflows is the other half. We propose two system-level components to support and harness our N-D LLM: **Semantic Tensor Memory** and **Auto-IB Orchestrator**. These act as the "memory and manager" around the core model, enabling long-term retention, self-optimization, and monitoring.

Semantic Tensor Memory (STM): This is a memory subsystem designed to store the model's knowledge and experiences in a structured, multi-dimensional form. Traditional LLMs often rely on their internal weights for long-term memory or use external vector databases to store embeddings of past interactions. Our STM concept is closer to a knowledge base that the model can write to and read from, but using the same multi-dimensional representations that our model operates with. Concretely, whenever the model processes an input and produces an output, the STM can log a **tensor record** that includes: - The compressed representation Z (or some projection of it) that the model generated for that input. - The output it gave (and perhaps the correctness feedback if available). - Metadata fields: time, context, any fields like source or user ID, etc.

Because Z is a compact summary of the salient info, storing Z for each interaction is far more efficient than storing the entire raw input. Over time, the STM becomes a repository of what the model found important across many instances. When a new query comes, we can consult STM to see if a similar context has been seen or if parts of the input have appeared before. For example, suppose a user asks a question about a document that the model has processed in the past – the STM might have a cached compressed representation or key facts from it, enabling faster or more accurate responses (like a learned retrieval cache).

The "Semantic" part implies we store not just opaque vectors but ideally interpretable entries. We might design the STM schema such that certain fields of Z are labeled. For instance, if part of the input includes an image and the model identified objects, we could store an entry like: [ImageID123, {"objects": ["dog", "park"], "caption": "A dog in a park"}, encoding_vector]. This blends symbolic and vector memory – something akin to a **long-term working memory** for an agent. In essence, STM could operate like a **scratchpad knowledge graph** that accumulates multi-modal facts the model has processed, each fact accompanied by the high-dimensional context embedding from which it was derived.

From a systems perspective, the STM would be implemented using a database or specialized store (could be a vector database for similarity search combined with a traditional database for structured info). The key innovation is storing multi-dimensional context: e.g., time series data might be stored indexed by time; spatial data by coordinates; text by semantic concepts. This is beyond typical vector DBs which just store one big vector per entry – here we might store a set of field-specific embeddings. A query to STM could then be multi-dimensional as well ("retrieve past entries that overlap in location AND have similar semantic embedding AND recent in time").

Use of STM: - *Memory Augmentation:* If the LLM is part of an agent that converses over time or does tasks iteratively, it can query STM for relevant past knowledge instead of relying purely on its finite context window. Because STM entries are compact, many can be retrieved and fed in if needed. - *Transparency and Logging:* Each decision the model makes can be traced to the STM entries it accessed or wrote. This helps in

debugging and explaining the model's behavior. For instance, if the model answered a question, we can show which memory entries (past cases or facts) were most similar to the current context. If a hallucination occurred, we might check STM to see if an incorrect memory was retrieved or if that info was never seen (so model made it up). - *Continual Learning:* Over time, the STM can be distilled into updated model weights or fine-tuning. The orchestrator might observe that a certain piece of knowledge is frequently accessed in STM, and decide to fine-tune the model on it to internalize it. Or, if new patterns emerge (like frequently dropped info in compression), it can adjust.

Auto-IB Orchestrator: This component functions as the high-level manager that continuously analyzes the model's performance and orchestrates improvements in the information bottleneck and input processing. It has several roles:

1. **Input Analysis and Routing:** Before the model processes an input, the orchestrator can predict how to handle it. For example, it might use heuristics or a lightweight ML model to estimate which fields of the input are most likely relevant. If the input is extremely large (say a 100-page PDF), the orchestrator could chunk it and only send one chunk at a time through the model, using IB compression to pick out the most relevant chunk first. This is similar to the idea of a director deciding how to allocate the model's attention across parts of input. It could even decide to skip processing some modalities if they seem irrelevant (e.g. if a question is purely textual, maybe don't run the heavy vision encoder on every page's image, unless needed). Essentially, it *predicts the bottleneck usage* – how many tokens likely needed, where to allocate them – before actually running the full model. This kind of meta-decision could be based on past experience (logged in STM). For instance, the orchestrator might recall that for a certain user's queries, images were never relevant, so it lowers priority of image field.

2. **Dataset Generation for Bottleneck Training:** The orchestrator can generate synthetic training data to continuously refine the compression. One approach is **self-supervised dataset creation**: take an instance (X, Y) that the model handled, then alter or stress it and retrain. For example, take a document and a question, remove the one paragraph that actually contains the answer, then feed it to the model – the model should ideally indicate "not answerable". If it instead tries to guess, the orchestrator flags this as a hallucination example, and generates a training instance where the correct output is "I don't have enough information". By doing this systematically, we build a dataset of "when to say I don't know". Another type: generate paraphrases or irrelevant inserts in input to test if compression still picks the right info – basically adversarial examples. The orchestrator could use a large language model (the LLM itself or another) to create these variations. For instance, ask the model to summarize the document, then insert random summary sentences back into the text to create distractors, then see if our bottleneck wrongly keeps them. These become training data points to teach the compression module to ignore distractors. This is akin to *hard example mining*: find cases where the model's bottleneck fails and fine-tune on them.

3. **Monitoring and Adaptive Tuning:** The orchestrator keeps track of metrics like average compression ratio, tasks where performance was suboptimal, distribution of token usage per field, etc. If it notices, say, that the model is consistently using nearly all available tokens for certain field but still not getting questions right, that might indicate that field requires a more powerful encoder or more capacity. Or if a field is rarely used, maybe the model or the orchestrator can choose to drop that field to save computation. These decisions could feed back into a system configuration (for instance, dynamically turn off processing high-res images if they aren't helping answer certain queries, thus

saving latency). The orchestrator essentially can reconfigure parts of the pipeline on the fly or between sessions.

4. **User Personalization & Privacy Controls:** If deployed in say a personal assistant context, the orchestrator could allow users to set policies like "Don't store my location data in memory" or "Always prioritize privacy: don't include exact names in memory, only abstract it." It can then enforce these by filtering fields before logging to STM or by instructing the compression module to exclude some fields from output (in extreme, dropping certain dimensions for compliance). Similarly, for bias control, orchestrator can monitor if certain types of content are always being filtered out and adjust thresholds to avoid unfair bias (for example, ensure that if a document is from a minority group, the salience model is not inadvertently underestimating it due to training bias).

Integration Flow: The orchestrator and STM work together around the model: - When a new query comes, orchestrator possibly retrieves related STM entries (like relevant knowledge). Those could be fed as an extra context field (like an "augmented prompt"). Because our model can handle N fields, adding a field like "memory_context" with a few tokens from STM is straightforward. It might even attach a provenance field indicating these are memory-derived. - The orchestrator then calls the field encoders, does compression, gets output. It monitors the compression: if it sees, for instance, that maximum allowed tokens were used and maybe truncated, it might decide to run another round with a different strategy (like, "the answer might be in part we truncated, let's compress that part separately and then answer"). This points to an iterative approach: the orchestrator can do multi-pass reading. Our architecture supports it because the model could output something like "The answer is likely in section 2. Please provide section 2 text." The orchestrator sees that and can comply. - After answer, the orchestrator stores any new insights to STM and possibly updates any models (the salience predictor, etc.) based on feedback.

Analogy to Human System 2: We call it Semantic *Tensor* Memory and not just memory to emphasize storing rich representations (tensors), akin to a human storing a multi-sensory memory of an event (sights, sounds, concepts). The orchestrator corresponds to a conscious deliberation layer – deciding where to focus, when to recall memories, when to double-check or gather more info. By formalizing these, we aim to push beyond the end-to-end black-box paradigm to a more **transparent and controllable LLM system**. This also resonates with the notion of "System 2 AI" where an AI can reason over structured knowledge and its own intermediate computations, not just do end-to-end neural processing [49] [50].

Implementing STM and the orchestrator would likely involve extending existing frameworks (like using a vector DB such as FAISS or Milvus for STM initial version, and writing high-level logic possibly with something like LangChain or custom code for orchestration). Importantly, these are not meant to involve training a giant model themselves; they are lighter-weight components orchestrating the main heavy model.

Benefits Recap: - STM ensures continuity (the model doesn't start from scratch every query) and provides a basis for explanation (by showing what info was retained/used). - Orchestrator ensures that the information bottleneck remains optimal as tasks vary, and that the system can adapt (either through meta-decisions or triggering retraining). It is like an **auto-tuner** for the model's IO: if a new type of input comes in that the model isn't compressing well, the orchestrator flags it and perhaps collects examples to fine-tune the model on those. - Both components are essential for real-world deployment because they handle concerns like performance scaling (orchestrator can manage multi-step processing for very large data) and safety

(memories can be audited for sensitive data, orchestrator can intervene if the model is about to output something unallowed by policy).

In summary, the system integration aspect acknowledges that an advanced LLM architecture alone isn't enough; we need a supportive ecosystem that logs, learns from, and guides the model's operation. This combination is what will ultimately yield a robust, *continually improving* AI system.

Ethical and Safety Considerations

Expanding LLMs to handle N-dimensional inputs and adaptive compression introduces new ethical and safety challenges alongside the technical benefits. We address these proactively:

Privacy of Input Fields: With richer inputs come greater risks of sensitive information exposure. Fields like geolocation, timestamps, user interaction data, or images can contain personally identifiable information (PII) or private context. For example, a user's location trajectory (time + geo fields) is highly sensitive. Our model and memory system must handle such data with care. This means: - Implementing **field-level access controls**: certain fields (marked sensitive) might be encrypted or omitted when logging to Semantic Tensor Memory. Perhaps only a hashed or coarse version is stored (e.g. round coordinates to city level, remove exact timestamps). - **User consent and transparency**: If an AI assistant uses these fields, the user should know (e.g. "Using your location to improve answer accuracy. Is that okay?"). Also, any stored memory involving user data should be governed by retention policies (like auto-deletion after some time, if required by privacy regulations). - When the model generates outputs, it should avoid unintentionally revealing private field data. For instance, if the model has access to a "user identity" field, we need to ensure it doesn't leak that in the answer unless explicitly allowed ("Based on your calendar, [Your Name] has a meeting..." might be fine if user asked, but leaking to other users is not). This ties into the orchestrator possibly *masking or stripping* certain fields when not needed, or the model being trained to not output certain fields unless prompted.

Bias in Routing and Compression: The model's compression decisions and multi-modal integration could inadvertently encode biases. For example, if the training data often marked certain types of documents or sources as "less important", the model might systematically drop or compress those more, leading to worse performance for those groups. Concretely, imagine a scenario: the model processes resumes with demographic info; if not carefully handled, it might learn to pay less attention to resumes from certain backgrounds (a serious fairness issue). Similarly, a salience model might be biased (say it highlights content from Western sources as more important than non-Western sources). To mitigate: - We include **diverse training data** for the compression module, so it doesn't learn a skewed notion of importance. This includes making sure that when learning to drop information, we aren't always dropping the same type of information (which might correlate with sensitive attributes). - Conduct **bias audits**: run the model on inputs that are identical except for a sensitive attribute and see if the output or compression differs. For example, two documents differing only in author name (one male, one female) – the processing and answer should remain the same. Or queries about different cultures – does the orchestrator retrieve memory differently? - The ethical statement from mpLLM warns that limited demographic data can yield models that underperform on underrepresented groups [51]. We take a similar stance: explicitly note where our training data lacks coverage and avoid over-claiming generality. Future work will involve incorporating fairness analysis, as they suggested [51]. - The MoE routing could also have biases: if an expert is specialized on data that mostly comes from a certain group, and the router wrongly or rightly directs certain demographics always to a particular expert, that could cause unequal treatment. We ensure *balance* by regularizing expert

usage (so one expert doesn't exclusively see one type of data unless intended and validated). - **Provenance fields** can be double-edged: while they help grounding, the model might over-rely on them ("if source is blog, ignore content" could be learned – sometimes good, but perhaps it should still consider content). We should monitor if provenance leads to systematic dismissal of, say, non-English content or minority voices. One guard could be to have the model justify: if it dropped something due to low trust, maybe flag that. At least in analysis, we can see if that happened.

Hallucination vs Accuracy Trade-off: Reducing hallucinations is a goal, but we must also be wary of *overcompression* where the model might omit or ignore subtle information, leading to confident but incomplete answers. There is a safety aspect: a hallucinated answer is bad, but an incomplete or out-of-date answer can also be problematic if the user relies on it (imagine an AI doctor missing a critical detail because it was compressed away). Our approach addresses hallucination by grounding, but we also need to ensure **calibration** – the model should express uncertainty when it's unsure due to compression. Training on out-of-scope queries (with an appropriate response like "I don't have that information") will be important. We likely will incorporate a special output token or mechanism for "not enough info", and ensure the model triggers it when relevant. This reduces the chance of a hallucination filling the gap.

Misuse of Multi-Dimensional Models: A more capable model that understands images, text, and metadata could be misused for surveillance or manipulation if not safeguarded. For example, such a model could potentially identify individuals in images combined with text (face recognition plus name tags). Our design doesn't inherently focus on face recognition, but it's a risk if integrated incorrectly. We must enforce the **image safety policies**: e.g. not outputting personal identities from images, as per the initial policy. If the model has a "person name" field from somewhere, it shouldn't blurt it out describing an image, unless that's explicitly allowed in context. We can implement filters to blank out certain outputs or to refrain from some types of multi-modal inferences (like a rule: do not combine geolocation and user identity to guess where someone lives – that's a privacy line). There's also the risk of model bias in interpreting images (vision models have had issues like racial bias in captions). Our training should include instructions/policies that align with safe behavior (e.g. not making sensitive attribute guesses from images, as per image policies "do not guess race or religion from image").

Robustness and Errors: Multi-modal systems can fail in new ways. For instance, if one modality is adversarial (imagine a malicious image that causes the vision encoder to output garbage), could that confuse the whole LLM into a wrong or harmful output? We should attempt some **adversarial testing**: provide conflicting info in text vs image and see what happens. Ideally, the model should either reconcile (maybe present both possibilities) or default to a safe failure. The orchestrator might detect inconsistencies (it could compare answer the model would give from text alone vs image alone – if they differ widely, flag the query for caution or ask clarifying questions). Also, the system being complex (with memory, orchestrator) means more points of failure. It's important to have **fallback behaviors**: e.g., if the orchestrator or memory fails, the model should still be able to do something reasonable (maybe revert to just using current input without fancy compression, albeit with performance loss, rather than crashing or giving nonsense).

Transparency and Explainability: With N-D input and a bottleneck, it could be non-transparent what the model actually used to answer. We argue our design *improves* transparency because the bottleneck Z is an explicit intermediate that could be inspected. For instance, one could decode or highlight which parts of input made it into Z. The attention scores or selected tokens in compression can be visualized, essentially giving an explanation: "The model focused on these sentences and this part of the image." We plan to use

that as a feature, but also caution: attention-based explanations can be imperfect. We will however log such info for developers or even end-users in critical settings. If the model is used in a high-stakes domain (medical, legal), having a trace of what info it used (maybe via highlighting the source document or recalling memory IDs) is valuable for trust and auditing.

Ethics of Memory: The Semantic Tensor Memory storing user data or any data means the system is **accumulating knowledge that could include sensitive or copyrighted info**. We must have policies for that: - If the AI reads a copyrighted document, storing it in memory (even compressed) might be problematic if later used to generate content. However, if compression truly distills only high-level info, maybe it's akin to "learning" from it which is fair use, but we have to be careful. One solution: mark entries in memory with usage rights and ensure the model doesn't output large verbatim text that was only in memory (unless allowed). - If a user asks something and then invokes the right to be forgotten, we need to remove their data from STM. Therefore, memory should be erasable by user request (which is easier if memory is separate from model weights – a plus of our design). - There's also the possibility of model picking up biases from user-specific memory (if a user often speaks a certain way, model might generalize incorrectly). But since memory is user-specific typically, this might not affect global model but could reflect back biases (like if a user had biased content, the assistant might reinforce that in conversation; handling that goes into content moderation territory which is another facet).

In implementing this ethically, we will consult guidelines like ones from EU on AI privacy [52] and existing bias mitigation research [53]. This includes making sure we test for biases (as described) and provide usage toggles (maybe a user can turn off certain sensors – e.g. "don't use my microphone input to anticipate my mood" – if such a field existed in future).

Finally, **safety in outputs**: Even if grounded, the model could be asked to do harmful things with multi-modal input (like "here's a floorplan, help me plan a break-in"). We need to apply the same content moderation filters as usual, which means scanning the prompt and perhaps the vision input for disallowed content. The orchestrator can integrate with a safety system: it might run a lightweight classifier on the combined input to detect if this is a request violating policies. If so, it stops and responds with a refusal. This is standard but must extend to images or other fields (like an image with extremist symbolism should be caught before the model comments on it in a disallowed way). Ensuring the model doesn't learn to bypass these when compressing is also key (it should not, for example, encode a hate symbol in Z in a way that the decoder then outputs hate speech – training data and hard constraints should prevent that).

In summary, while our advanced architecture opens new possibilities, we embed ethics at every layer: from training (diverse data, IB to reduce misinformation spread) to inference (privacy filters, safety checks) to memory (user control, transparency). By design, many of our choices (like encouraging the model to abstain when unsure, grounding answers in input, logging what it used) inherently improve safety compared to a vanilla LLM that might confidently make up answers. The multi-dimensional grounding helps ensure the model's outputs have a traceable origin, aligning with goals of responsible AI that can *explain its reasoning* and *refuse when it lacks sufficient basis*. We will document limitations and ensure that any deployment of this system is preceded by thorough testing under an ethical framework.

Conclusion and Roadmap

We have outlined a vision for **information-efficient, grounded large language models** that break free from the limitations of 1-D token streams. By marrying principles from information theory with innovations

in multi-modal architecture, we can build LLMs that *do more with less*: extracting maximal knowledge from each input token while remaining faithful to the input evidence. This approach is a step toward AI systems that are not just larger, but smarter in how they use information – a crucial attribute on the path to general intelligence.

Summary of Contributions: - We surveyed the evolution from 1-D to 2-D LLMs, learning how spatial layouts and visual encoders (as in DeepSeek-OCR and LayoutLM) dramatically improve context handling [5] [9]. Building on their successes and limitations [24] [28], we generalized to N-D inputs, defining a taxonomy of dimensions (space, time, depth, etc.) and analyzing their effects on reducing uncertainty and hallucination. - We introduced a novel architecture featuring a **variable-rate token bottleneck**, guided by the Information Bottleneck theory [10] and realized via mechanisms like adaptive token selection and MoE routing [14]. This bottleneck ensures the model's internal context focuses on what truly matters for the task, dynamically adjusting compression level per input [30]. - We provided an implementation blueprint, detailing modules for field-specific encoding, multi-dimensional embedding, and integration with a decoder. We described how to train this system with a multi-objective loss (task performance + compression fidelity), leveraging both existing datasets and orchestrator-generated data. This serves as a practical roadmap for engineers to build a prototype system. - We outlined an extensive evaluation plan, including domain-specific benchmarks and controlled experiments, to rigorously quantify the benefits. By comparing token-for-token, we will demonstrate the efficiency gains – for example, matching SOTA OCR accuracy with 10× fewer tokens [6], or outperforming text-only baselines on layout-intensive tasks using equal context size [54]. We will also validate improved grounding via reduced hallucination rates [7]. - We integrated system-level components (Semantic Tensor Memory and Auto-IB Orchestrator) to manage long-term knowledge and adaptivity. These ensure the model continues to learn from experience, remains interpretable, and can be tuned on the fly as new requirements emerge. - Throughout, we addressed ethical and safety aspects, ensuring the design promotes responsible AI use – by guarding privacy of additional fields, striving for fairness in what information is retained or dropped, and enabling the model to say "I don't know" rather than fabricate answers when input is lacking.

Roadmap for Development: 1. **Prototype Phase (Next 3–6 months):** Implement the core architecture with 2-D inputs (text + layout as a starting point) and the token bottleneck. Conduct initial experiments on document QA and long text summarization. This will yield early results proving the viability of learned compression (e.g. replicating DeepSeek's optical compression with our architecture). Simultaneously, implement the basic orchestrator logic for context filtering and set up the Semantic Memory schema (even if just as a logging DB initially). 2. **Expand Modalities (6–12 months):** Incorporate image inputs and possibly simple temporal data. This would involve training on multimodal datasets like DocVQA (for images+text) and evaluating on tasks like VQA. Introduce additional fields like "salience" by using off-the-shelf models to generate importance scores for training (e.g. use a question-focused summarizer to label which sentences are important). Also integrate a pre-trained vision model to jump-start image understanding. By this stage, we aim to show that adding image and other fields indeed improves accuracy on tasks like "describe this document" or "answer questions about this diagram" with fewer tokens than a text-only approach. 3. **Generalize N-D (12–18 months):** Push to more dimensions: possibly video (temporal sequences of images) or audio (time + frequency), or structured data (text + database tables). This will test the scalability of our approach. We might need to optimize the compression module further (e.g. hierarchical compression for video frames) and refine the MoE routing as dimensions grow. During this phase, heavy benchmarking occurs to compare with specialized models: e.g. compare to a dedicated video QA model or an audio-transcription model to see if our unified approach can match them when given the same modalities. 4. **Memory & Feedback Integration (18–24 months):** Turn on the Semantic Tensor

Memory for real use cases. For instance, deploy the system in a closed-loop environment (maybe as an assistant on a static knowledge base, or a document processing workflow) where it can accumulate knowledge and re-use it. Evaluate benefits like: does memory improve response time or quality over time? Implement the feedback loop where the orchestrator uses memory to fine-tune the model periodically (akin to continuous learning, but carefully to avoid catastrophic forgetting or privacy leaks). We will also flesh out interpretability tools – e.g. a UI that highlights what info was retained/dropped, to build trust with users. 5. **Robustness and Safety Testing (24+ months):** Before broader deployment, conduct extensive adversarial and bias testing as discussed. Work with domain experts for high-stakes fields (medical, legal) to validate that the model's compression isn't omitting critical details and that its grounded outputs meet professional standards. Address any issues via targeted data augmentation or rule-based fixes via the orchestrator. Also engage with ethicists or regulatory experts to ensure compliance with data handling, especially if the system handles personal data in memory. 6. **Deployment and Monitoring:** Finally, deploy in target applications – perhaps as a backend for summarizing long reports, a multimodal chatbot, or an analysis tool for complex data. Monitor real-world performance and gather user feedback. The orchestrator's telemetry will be crucial here: it might catch if the model starts to struggle with a new kind of input, and we can quickly collect those cases to retrain or adjust.

Long-Term Vision: If successful, this approach paves the way for AI systems that can **fluidly integrate diverse information** – textual, visual, numerical, contextual – and do so in a way that is both **efficient (computationally and information-theoretically)** and **trustworthy**. Rather than brute-forcing larger context windows or parameter counts, we focus on *intelligent information processing*, which is more scalable in the long run (akin to how humans don't read every word of a textbook with equal focus; we highlight, skim, and recall relevant pieces). This is a stepping stone to more general intelligence: an AI that can handle a breadth of input types, distill what matters, learn from each interaction, and not be tripped up by irrelevant distractions or its own training priors when faced with new situations.

In conclusion, by grounding LLMs in the multi-dimensional reality of data and enforcing a disciplined bottleneck on what they "perceive", we aim to create models that are **both knowledgeable and self-aware of their knowledge limits**. They will know what they know (and highlight why, with sources or memory), and know what they don't know (and gracefully ask for more info or abstain, rather than hallucinate). Achieving this will significantly increase the reliability and utility of AI systems in complex, information-rich domains, bringing us closer to truly assistive and general AI.

Acknowledgements: (In a real whitepaper, we might thank collaborators or cite the works we heavily drew upon, like Tishby's IB theory [10], Nagle et al.'s rate-distortion framework [11], DeepSeek team for inspiration [8] [5], etc. But since this is an internal document style here, we have cited inline as required.)

---

1 2 10 29 31 32 43 QUITO-X: An Information Bottleneck-based Compression Algorithm with Cross-Attention
https://arxiv.org/html/2408.10497v1

3 4 Why large language models struggle with long contexts
https://www.understandingai.org/p/why-large-language-models-struggle

5 6 8 16 21 22 23 28 38 41 DeepSeek OCR is here. How to use DeepSeek OCR for free? | by Mehul Gupta | Data Science in Your Pocket | Oct, 2025 | Medium
https://medium.com/data-science-in-your-pocket/deepseek-ocr-is-here-37096b562bb0

7 13 14 17 39 40 42 51 A Multimodal LLM Approach for Visual Question Answering on Multiparametric 3D Brain MRI
https://arxiv.org/html/2509.25889v1

9 15 18 19 20 Your ultimate guide to understanding LayoutLM
https://nanonets.com/blog/layoutlm-explained/

11 12 30 Fundamental Limits of Prompt Compression: A Rate-Distortion Framework for Black-Box Language Models | OpenReview
https://openreview.net/forum?id=TeBKVfhP2M

24 25 26 27 46 54 A Bounding Box is Worth One Token: Interleaving Layout and Text in a Large Language Model for Document Understanding
https://arxiv.org/html/2407.01976v1

33 34 35 36 37 Bottlenecked Transformers: Periodic KV Cache Consolidation for Generalised Reasoning | OpenReview
https://openreview.net/forum?id=fWgKnl4itC

44 The User-Centric Geo-Experience: An LLM-Powered Framework for ...
https://arxiv.org/html/2507.06993v1

45 Large Spatial Models (LSM)
https://giscience.psu.edu/2025/05/02/building-the-large-spatial-models-lsm/

47 GitHub - deepseek-ai/DeepSeek-OCR: Contexts Optical Compression
https://github.com/deepseek-ai/DeepSeek-OCR

48 SpatialLLM: From Multi-modality Data to Urban Spatial Intelligence
https://arxiv.org/html/2505.12703v1

49 50 Semantic Tensor Memory Framework for System 2 Generative AI | by Jack Brzezinski | Oct, 2025 | Medium
https://medium.com/@brzezinski_j/semantic-tensor-memory-framework-for-system-2-generative-ai-299fd77e3f18

52 [PDF] AI Privacy Risks & Mitigations – Large Language Models (LLMs)
https://www.edpb.europa.eu/system/files/2025-04/ai-privacy-risks-and-mitigations-in-llms.pdf

53 Bias in Large Language Models: Origin, Evaluation, and Mitigation
https://arxiv.org/html/2411.10915v1