Developer-Centric Adaptive Routing (DCAR) and Visible Expert Orchestration (VEO): An Efficiency-Oriented Architectural Framework for Foundation Model Deployment

Author: Faisal Hassan, Independent Systems Architect

Abstract

Foundation models — large-scale pretrained Transformer networks (e.g., GPT-3) — have demonstrated unprecedented generality in language understanding and generation [1]. However, invoking these monolithic models for narrow developer tasks often wastes computational resources: a tiny prompt like "write a Dockerfile" may call a 100-billion-parameter model, incurring unnecessary token usage, latency, and energy costs. We propose a **Developer-Centric Adaptive Routing (DCAR)** framework introducing a controllable **Visible Expert Orchestration (VEO)** layer between user requests and foundation models. DCAR uses lightweight analyzers and confidence-scoring policies to route queries only to specialist "expert" models (e.g., a Python-coding model, a Docker-model, a cloud-infra model) when needed. Importantly, the routing is exposed to the developer: a user can manually select or approve experts (or enable auto-selection) for each task, thereby governing cost and precision. Unlike internal Mixture-of-Experts (MoE) techniques, VEO makes the orchestration visible and configurable. We outline a concrete DCAR architecture, define metrics for token efficiency, latency, and energy use, and describe an experimental methodology to simulate DCAR vs. monolithic baselines. We hypothesize that DCAR will significantly reduce inference resource usage and cost for common development workflows without sacrificing answer quality. This work reframes AI system optimization from scaling up monolithic models to efficiency-centric model orchestration.

1. Introduction

In recent years **Artificial Intelligence (AI)** has rapidly progressed from narrow, rule-based systems toward statistical machine learning (ML) and now *deep learning* (DL) with multi-layer neural networks [2]. Neural models have particularly advanced natural language processing (NLP) tasks: early sequence models like LSTMs gave way to Transformer architectures [2] [3]. The 2017 landmark "Attention Is All You Need" paper introduced the Transformer, a neural network using multi-head self-attention to process sequences in parallel [2] [4]. Transformers enabled very large models that could be pretrained on massive text corpora. These **foundation models** (trained by self-supervision on broad data) can be fine-tuned or prompted for many downstream tasks [1]. Notable examples include BERT [3] (an encoder-only Transformer for language understanding) and the GPT family of decoder-only models (GPT-1 in 2018, GPT-2 in 2019, GPT-3 in 2020, and GPT-4 in 2023 [3] [5]). The 2022 release of ChatGPT (a GPT-3.5-based chatbot) and GPT-4 (a multimodal model) generated widespread public interest, making large language models (LLMs) central to modern AI applications [5].

This evolution has brought extraordinary capabilities: LLMs can generate coherent code, summarize documents, answer questions, and even converse in natural language. However, these gains come at an extreme computational cost. As Strubell *et al.* (2019) note, large neural NLP models require "exceptionally large computational resources" to train, with correspondingly large financial and *environmental* costs [6]. While most attention has focused on training cost, inference cost is also significant: every time a model processes tokens, it consumes GPU/TPU compute, electricity, and often money (via API pricing). For example, an LLM call may use thousands of gigaflops of compute; cumulatively across many queries, this yields a nontrivial carbon footprint [6] [7]. Indeed, recent analyses highlight that even inference and storage contribute to the carbon footprint of LLMs [7]. From a business perspective, enterprises report that naïvely calling LLM APIs leads to unpredictable costs and governance challenges: **"costs scale with tokens rather than requests," and early implementations "run into unpredictable inference costs, limited visibility into usage, [and] provider lock-in"** [8]. Consequently, companies are beginning to build "AI gateways" and orchestration layers to control and optimize LLM usage [9].

In typical developer workflows today, most LLM queries are **narrow and domain-specific** (e.g., "write a unit test in Python," "convert YAML to JSON," "generate a Dockerfile," "explain this SQL error," etc.). Yet common practice is to send all such queries to a large general-purpose model (e.g., GPT-4). This monolithic approach **over-provisions** compute: for a simple coding question, why pay for 100B parameters if a 1B-parameter code-focused model could suffice? To draw an analogy, it is like firing up a supercomputer to calculate 2+2. We argue that this inefficiency is widespread in developer settings. Moreover, large LLMs often embed specialized knowledge (e.g., coding or math skills) unevenly: some "experts" (subnetworks) are better at certain domains [10]. Recent architectures like Sparse Mixture-of-Experts (MoE) exploit this by routing token vectors through a subset of expert sub-models [10]. However, MoE routing is internal to the model and opaque to users; developers cannot easily control or even see which experts are used.

Contributions: In this work, we propose an architectural paradigm shift for LLM deployment in developer tools. Rather than improving models in isolation, we focus on **inference orchestration**: dynamically routing queries to specialized models with explicit developer control. Our **Developer-Centric Adaptive Routing (DCAR)** framework introduces a visible orchestration layer between the user and the models. DCAR performs an initial analysis of each request (identifying its domain(s), complexity, and constraints), then applies a **Visible Expert Orchestration (VEO)** policy to decide which underlying models to invoke. For example, a query may be tagged as "Python coding" and "DevOps," triggering calls only to a Python-model, a Docker-model, and a cloud-infra model. Crucially, DCAR allows manual override or confirmation: in "manual mode" the developer selects the domains; in "auto mode," DCAR suggests models but can ask for approval.

Our approach contrasts with current practice. Instead of a single API call to a fixed LLM, DCAR can involve multiple smaller calls, each tightly scoped to the task. By exposing the orchestration, developers gain cost transparency and control. We outline how to implement DCAR as middleware or an API gateway, define metrics (token count, latency, energy, cost), and propose an experiment design comparing DCAR against monolithic LLM invocation. We hypothesize that for typical software-development prompts, DCAR will reduce token use and energy roughly proportional to the number of experts engaged, with negligible accuracy loss. Overall, this work reframes LLM system design: moving from brute-force model scaling toward **efficient orchestration**.

The rest of the paper is structured as follows. In Section 2 we review AI foundations (ML, DL, neural nets, tokenization, attention, transformers, and LLM history). Section 3 discusses related work on foundation

models, mixture-of-experts, and inference optimization. Section 4 precisely states the problem of over-provisioned inference in developer workflows. Section 5 presents the DCAR/VEO framework and its components. Section 6 describes an experimental methodology to evaluate efficiency. Section 7 outlines expected evaluation metrics and qualitative analysis. Section 8 discusses trade-offs and practical considerations. Section 9 sketches an incremental adoption strategy for enterprises. Section 10 acknowledges limitations. Section 11 addresses broader impacts on energy and sustainability. We conclude in Section 12 by summarizing our vision for orchestrated AI systems. A Glossary (Section 13) defines key terms used throughout.

2. Foundations of Artificial Intelligence

To set the stage, we briefly review key concepts in AI, machine learning, deep learning, and neural network models. Our goal is to ensure the paper is self-contained for readers of varied backgrounds, while citing authoritative sources.

- **Artificial Intelligence (AI):** Broadly, AI refers to computational systems that perform tasks which would require "intelligence" if done by humans [11]. A classic definition (McCarthy, 1955) is "the science and engineering of making intelligent machines" [12]. Modern AI emphasizes learning from data rather than hard-coded rules.
- **Machine Learning (ML):** ML is the subfield of AI focused on algorithms that improve through data and experience [13]. In supervised learning, models learn to predict labels from examples; in unsupervised or self-supervised learning, models discover structure from unlabeled data; in reinforcement learning, agents learn sequences of actions to maximize reward [14].

- **Deep Learning (DL):** DL uses deep (multi-layer) neural networks with millions or billions of parameters [2]. Such networks can learn hierarchical representations of data. For instance, in vision, early layers learn edges, later layers compose them into objects. A key recent innovation in DL is the **Transformer** architecture [2].

- **Neural Networks:** An artificial neural network is a parameterized function, loosely inspired by biological neurons. It consists of layers of weighted sums and nonlinear activations. Traditional DL networks include feedforward networks, convolutional nets, and recurrent nets. The Transformer is a specific neural architecture for sequences, introduced in 2017 [2].

- **Data and Tokenization:** In NLP, raw text must be converted to numbers for neural networks. The first step is *tokenization*, which splits text into discrete units (tokens) such as words or subwords [15]. Tokenization can be at the word level, character level, or subword level (e.g., Byte-Pair Encoding or WordPiece) [16] [17]. For example, the sentence "Chatbots are helpful." can be tokenized into words `["Chatbots","are","helpful","."]` or into subwords like `["Chat", "bot", "s", "are", ...]`. Each token is then mapped to a unique integer index.

- **Embeddings:** Each token index is associated with a vector embedding, a fixed-length real-valued vector that encodes semantic information about the token [18]. Word embeddings capture word meaning as positions in high-dimensional space: similar words have nearby embeddings [18]. Embeddings may be learned jointly with the model (as in Transformers) or precomputed (as in Word2Vec or GloVe) [18]. In practice, modern NLP systems use large learned embedding tables.

- **Attention Mechanism:** The attention mechanism allows a model to weigh the influence of different tokens when processing a sequence. In self-attention, each token produces query, key, and value vectors; attention computes weighted sums of values where weights are based on similarity of queries to keys. Importantly, attention enables the model to incorporate *context* flexibly, without fixed-size hidden states. Multi-head attention splits each query/key/value into several subspaces (heads) and performs parallel attention in each head [4]. This lets the model capture diverse aspects of token relationships simultaneously. The outputs of the heads are concatenated and linearly transformed [4]. The Transformer uses multi-head self-attention as its core operation, allowing it to process sequences with long-range dependencies and parallel computation [2] [4].

- **Transformer Architecture:** A Transformer model typically consists of an **encoder** stack and a **decoder** stack (the original "Attention Is All You Need" design [3]). Each encoder layer has a multi-head self-attention sublayer followed by a feed-forward network; decoder layers add an additional encoder-decoder attention sublayer. There are also simpler "encoder-only" Transformers (like BERT) and "decoder-only" Transformers (like GPT) [3]. In all cases, input text is tokenized and embedded, and positional encodings (e.g., sinusoids) are added to embeddings to represent order [19]. The encoder processes the whole input sequence in parallel; the decoder generates output tokens iteratively, attending to both prior outputs and the encoder outputs.

The key feature of Transformers is parallelism: unlike RNNs, they do not process tokens strictly sequentially. This enabled training on large datasets with modern accelerators. Since 2017, Transformers have become ubiquitous in NLP (and beyond) [2] [3].

- **Vector Databases:** Vector embeddings can also be stored and queried outside a model. A **vector database** is a specialized storage system optimized for high-dimensional vectors [20]. It allows efficient similarity search: given a query embedding, the database returns the nearest stored embeddings (and their associated data). Vector databases (e.g., FAISS, Milvus, Pinecone) support indexing, metadata filtering, and scalable retrieval [20]. They are commonly used for retrieval-augmented generation: e.g., storing document embeddings for knowledge retrieval or long-term memory.

- **GPT and LLM History:** The history of large language models is marked by rapid scaling. After the 2017 introduction of the Transformer [3], researchers developed first large pretrained encoders and decoders. BERT (2018) popularized masked language modeling [3]. OpenAI's GPT-1 (117M parameters, 2018) showed that unsupervised pretraining improved many tasks [5]. GPT-2 (2019) scaled to 1.5B parameters and gained attention for text generation [5]. GPT-3 (2020) at 175B parameters demonstrated "few-shot" learning capabilities. OpenAI's ChatGPT (2022, based on GPT-3.5) and GPT-4 (2023) further showcased powerful generation and reasoning [5]. At the same time, open-weight models like Meta's LLaMA (2023), Mistral, DeepSeek (2025) and others emerged, offering alternatives to closed models [21]. Nearly all modern top-performing LLMs are built on Transformer variants [22].

In summary, modern AI systems use deep neural networks with attention to handle tasks that involve text (and now images, audio, etc.). Tokens are converted to embeddings and processed by layers of multi-head attention. Foundation models like GPT-4 are enormous Transformer decoders pretrained on massive data, enabling them to perform many tasks with prompting. However, the size and cost of these models raise practical challenges, especially when used as back-end engines for developer applications.

3. Related Work

We survey prior work relevant to our framework. We focus on: (a) key foundational model papers; (b) model architecture innovations (attention, MoE); and (c) systems-level ideas for optimizing inference and developer usage.

- **Attention and Transformers:** The seminal 2017 paper by Vaswani *et al.* **"Attention Is All You Need"** introduced the Transformer model for machine translation [3]. It showed that replacing recurrent networks with self-attention (including multi-head attention) achieved state-of-the-art performance while enabling full parallelization. This work paved the way for all subsequent Transformer-based models. Similarly, the rise of encoder-only models (BERT, 2018 [3]) and decoder-only models (GPT series) built directly on this architecture. Extensions to the Transformer (e.g. ALiBi, RoPE positional encodings, sparse attention variants) have been widely studied (see [14] or [20] for details).

- **Large Language Models:** OpenAI's GPT papers demonstrated scaling up Transformers to language modeling at unprecedented sizes (GPT-1 [5], GPT-2 [5], GPT-3 [5]). The GPT-3 paper (Brown *et al.*, 2020) showed that very large models enabled few-shot and in-context learning. Claude (Anthropic) and Google's Gemini have been proposed as similarly capable LLMs. However, all these models are monolithic decoders that treat any input uniformly. Other lines of work include retrieval-augmented models (RAG), instruction-tuned models, and system-2 style models (e.g. chain-of-thought LLMs [23]).

- **Mixture-of-Experts (MoE):** MoE architectures introduce sparsity by having *multiple sub-networks (experts)*, with a routing mechanism that selects a subset for each input. Early MoE work (e.g., "Sparsely-Gated Mixture of Experts" by Shazeer *et al.*, 2017) showed significant gains for large models. Recent large-scale MoE models (e.g., Google's GShard, Switch Transformer; or Mixtral, DeepSeekMoE in 2024-2025) use MoE layers where only a few experts (via Top-k gating) are active per token. The basic idea is: replace each Transformer feed-forward sublayer with *N* parallel feed-forward networks (experts) and a small gating network that assigns inputs to experts [10]. This allows scaling model capacity (billions of parameters) with minimal extra compute per token. Our DCAR framework is akin to MoE but *at the system level*: instead of internal routing, we route at the API layer. We expose the equivalent of the gating decisions to the developer, rather than hiding it inside a single model. This differs from prior MoE research which focuses on maximizing performance of one large model [10]; we focus on efficiency and control in multi-model inference.

- **Inference Optimization:** Some research addresses inference cost indirectly. Techniques include model quantization, distillation, caching, early exit, and adaptive computation. For example, smaller "student" models can mimic larger ones to reduce cost. In multi-turn systems, caching partial computation (e.g., Transformer key-value caches) speeds up generation. Other work proposes conditional computation (run only part of a model for a given input). However, these usually assume a single model. Recently, "dynamic routing" ideas have appeared in blog posts and concept papers. For instance, some practitioners suggest *cascaded inference*: try a cheaper model first, then escalate to a bigger model if confidence is low (as in [35] "Pattern 2: Cascading Intelligence"). Others discuss *parallel model querying* or *consensus* (as in [35] Pattern 3). These ideas align with DCAR's philosophy, but to our knowledge no formal system has been published that implements developer-selectable multi-model orchestration.

- **AI Infrastructure and LLMOps:** Commercial and open-source platforms (often called **AI Gateways** or **Model Orchestrators**) are emerging to manage LLMs in production. For example, the TrueFoundry blog [34] describes an AI gateway layer to handle cost, routing, retries, and observability for LLMs. Cloud providers and startups offer API proxies or multi-model endpoints. These systems acknowledge token-based billing and latency as key challenges, and propose routing logic and caching as solutions. However, most focus on enterprise usage (scaling dozens of applications) rather than developer workflows. Importantly, existing tools typically leave policy configuration implicit or static, whereas DCAR envisions fine-grained developer-driven policies (manual or automatic) on each request.

- **Energy and Carbon Footprint:** Several works highlight the environmental impact of large models. Strubell *et al.* (2019) quantify the CO_2 emissions of training NLP models, showing that training GPT-2 emits as much carbon as multiple cars over their lifetimes [6]. More recent work (Faiz *et al.*, 2023) models the *inference* carbon footprint of LLMs, noting that inference and operational costs add to the total emissions [7]. We build on these motivations: by reducing wasted inference, DCAR can reduce operational energy use.

- **Human-in-the-Loop and Interactive AI:** Our proposal aligns with "human-centered AI" principles [24]. By making expert selection explicit and giving developers control, we support transparency and user trust. In this sense, DCAR can be seen as a form of human-in-the-loop system design, where the user guides the AI pipeline.

In summary, the existing literature spans model architectures and system design, but there is a gap in formally combining these ideas for *developer-centric* efficiency. Our work draws on the strengths of MoE and multi-model orchestration, but reframes them with explicit developer governance. To our knowledge, DCAR and VEO are novel contributions to LLM deployment strategy.

4. Problem Definition

Large foundation models excel at general tasks, but their **monolithic deployment** creates inefficiencies in practice. We define the core problem as follows:

1. **Computational Over-Provisioning:** In developer workflows, most tasks are specific (e.g., coding, configuration, math) and do not require full generality. Yet the common approach is to call a single, very large model (e.g., GPT-4) for all tasks. This wastes computation. Each token processed uses all layers of a ~100B-parameter model, even when only a tiny subset of the model's knowledge is needed.

2. **Token and Energy Waste:** Every API call generates input and output tokens, each of which incurs cost (e.g., $/1K tokens) and energy (GPU cycles). Unnecessary tokens (e.g., boilerplate language) and unnecessary model scale drive up billing and power use. The per-token cost can be significant in high-volume settings; for example, at $0.03 per 1K tokens, answering 10000 small dev questions could cost hundreds of dollars and thousands of kilowatt-hours.

3. **Monolithic Latency:** A single large model often has variable latency depending on load and context. Developers often need quick responses (for coding assistance), but invoking the heaviest model for

simple queries adds latency. Moreover, long outputs (explanations or logs) can stretch context windows, slowing generation further.

4. **Lack of Visibility and Control:** Current systems treat the foundation model as a "black box" resource. A developer cannot see *why* a model gave a certain answer, nor adjust the reasoning path. Advanced users have no easy way to leverage smaller specialized tools or control costs on a per-query basis.

5. **Risk of Model Inefficiencies:** Even within a large model, the internal routing (if any) is fixed. Sparse Mixture-of-Experts in models like DeepSeekMoE or Mixtral automatically route tokens to experts, but this happens inside the model without user input. The developer cannot choose to bypass an expensive expert or insert a different one.

Example Use Case: Imagine a developer using an AI tool to generate code and documentation. She asks: "Write a Python function to sort a list." Under monolithic deployment, this question is sent to GPT-4. GPT-4 uses general world knowledge, context, and reasoning to answer — but essentially all that's needed is a known Python sorting snippet. A smaller Python-specialized model (or even a cached answer) would suffice. Yet the system blindly incurs GPT-4's cost. If later she asks "How to configure this in Docker?", again GPT-4 is used, even though the knowledge lies in a different domain (DevOps).

This pattern repeats for hundreds of questions: each time, a near-full-scale model is used. Over time, the cumulative waste is large. In contrast, if the system could detect the query's domain (Python vs. Docker vs. Cloud) and route to appropriate experts, it would **tailor compute to need**.

Formal Problem Statement: Let D be the space of developer prompts (queries) and M a set of available models of varying specialties and sizes. Traditional inference uses a single model $m_0 \in M$ (the largest) for all $d \in D$. We seek a routing function $R: D \rightarrow P(M)$ (where P(M) is a subset of models) such that for each query d, only the relevant models $\{m_1, m_2, \ldots\}$ are invoked. The goals are: minimize total token count (inputs+outputs), inference time, and energy, subject to maintaining or improving answer quality and developer satisfaction. Additionally, developers should be able to override or influence R(d) manually.

Desiderata: A practical solution must balance multiple factors:

- *Efficiency:* Dramatically reduce token usage and compute relative to monolithic baselines.
- *Quality:* Preserve correctness, coherence, and usefulness of answers. If multiple experts contribute, their outputs must be combined logically.
- *Latency:* Maintain fast response time; potentially benefit from parallel expert calls.
- *Cost-Awareness:* Allow explicit control of budget (e.g. always use cheap mode).
- *Transparency:* Expose routing decisions to the user for audit and control.
- *Feasibility:* Build on existing models/APIs with minimal architectural changes.

Current mainstream LLM platforms do not meet these criteria. The rest of this paper proposes and evaluates a new architecture to address them.

5. Proposed Framework: DCAR with VEO

We propose **Developer-Centric Adaptive Routing (DCAR)**, an inference-layer architecture that adaptively routes queries to specialized expert models. The key idea is to insert a **visible orchestration layer** between the developer's request and the underlying foundation models. Figure 1 illustrates the high-level workflow:

```
User Query → Analyzer → Routing Decision → (Expert Models) → Aggregator → Final Answer
```

Figure 1: *High-level DCAR workflow (conceptual)*. The **Analyzer** tags the query (e.g., domains, intent). The **Router** selects which expert models to call. The **Experts** (specialized LLMs) generate partial results. The **Aggregator** combines them into the final output. Developers can inspect and adjust routing.

5.1. Core Components

DCAR consists of the following modular components:

- **Request Analyzer:** A lightweight model or heuristic that inspects the user's prompt and any metadata (such as labels or context). It may perform domain classification (e.g., *"code", "math", "cloud", "documentation"*), complexity estimation, and constraint detection. For instance, the analyzer might use keyword triggers (e.g., presence of "import" or "function()" to tag "Python") or a small text classifier to identify relevant domains. It may also estimate confidence in simple answers (e.g., by querying a small model or using rule-based logic).

- **Domain/Expert Registry:** A configuration listing available expert models and their specialties. For example: `{ "Python": Model_Py, "Docker": Model_Dock, "Cloud": Model_Cloud, "SQL": Model_SQL, … }`. Each entry includes metadata like the model's cost (tokens per call), latency profile, and usage quotas. This registry is maintained by the developer or platform and may be updated as new expert models become available.

- **Routing Policy Engine (VEO):** The central decision logic. Given the Analyzer's tags and the registry, the policy engine decides which experts to invoke. Policies may be:

- **Manual Mode:** The system presents detected domains to the user for selection. Only user-approved experts are used.
- **Automatic Mode:** A rule-based or learned policy automatically selects experts. For example, if the query mentions "python" and "docker", pick Model_Py and Model_Dock. The policy may also consider cost constraints or latency budgets.
- **Hybrid:** The system suggests a routing but awaits user confirmation before calling expensive models.

The policy can also incorporate confidence thresholds. For instance, if a cheap expert returns a very high-confidence answer, skip calling more expensive ones (cascading strategy). Or, if combined domain tags suggest a multi-step task, plan a sequence of expert calls.

- **Expert Models:** A set of specialized LLMs, each fine-tuned or prompted for a specific domain. Examples include:
- A **Code Model** (e.g., CodeLlama) that excels at Python/Java/C++ code generation.
- A **DevOps Model** that knows Docker, Kubernetes, CI/CD syntax.
- A **Database Model** for SQL queries.
- A **Documentation/General Model** for writing explanations or summarizing.

- A **Math Model** for symbolic math or reasoning.
  These models can be smaller in size (e.g., 1B–10B parameters) since they focus on a narrow domain. They may be open-source or API-based. Importantly, they are *differentiated by their input prompt engineering or fine-tuning* (e.g., specialized system prompts).

- **Aggregator (Response Merger):** After experts respond, their outputs are combined. This could be as simple as concatenating answers if sub-tasks were independent, or more complex merging. In some cases, an additional pass of a meta-model may be used to blend responses ("resolve conflicts or redundancies"). For example, if the query involved both code and documentation, the Aggregator may format the final answer to include both parts. The aggregation logic should preserve coherence and adhere to the user's format requirements.

- **Dialogue/Feedback Loop:** (Optional) The system may iteratively refine. For instance, if an expert's answer is incomplete, the Router might add another expert or re-run an expert with follow-up context. The user could also ask follow-up questions which the Analyzer treats as new queries with their own routing.

5.2. Key Features and Workflow

1. **Query Tagging:** When the user submits a prompt, the Analyzer first tags it. Example tags could include domain keywords (`python` , `docker`), type (`explain` , `generate code` , `configure`), or urgency (`quick` , `detailed`). This can be done via a small ML classifier or simple pattern-matching. (E.g., a regex finds " `def` " to tag `Python` .)
2. **Domain Selection (VEO):** The tags are presented to the user. For example: *"Detected: [Python, Docker]. Select experts: [✓Python] [✓Docker] [ ]GCP."* The user can adjust these checkboxes. If in auto-mode, this step can be skipped (all detected tags are auto-selected).
3. **Model Invocation:** For each selected tag, the system calls the corresponding expert model. It feeds the original query (or a reformulated sub-query) to each expert. If the query requires sequential tasks, the system may call experts in sequence (e.g., first get code from Model_Py, then ask Model_Cloud how to deploy that code).
4. **Result Merging:** The raw outputs from the experts are merged. If multiple models answered the same aspect, a tie-breaking rule or an aggregator LLM could pick or synthesize a best answer. Often, each model handles its own piece (e.g., code vs doc vs infra), so merging is assembling a document with sections.

5. **User Response:** The final merged answer is presented to the user. Crucially, the interface can also show *which experts were called* and metrics (e.g., tokens used, cost). This transparency lets the developer refine the approach (e.g., deselect a costly model next time).

5.3. Example Workflow

To illustrate, consider an example prompt and DCAR flow:

- **User prompt:** "Write a Python function to upload an image to AWS S3, then provide an explanation of how it works."
- **Analyzer tags:** `Python`, `AWS S3`.
- **User review (manual mode):** Shows [✓Python, ✓Cloud (AWS S3)] selected.
- **Routing:**
- Call *Model_Python* with prompt *"Write a Python function to upload an image to AWS S3."* Model_Python generates a code snippet (e.g., using boto3) and returns the function code.
- Call *Model_Cloud* (tuned to AWS topics) with prompt *"Explain how the above Python function uploads an image to AWS S3."* Model_Cloud returns a natural-language explanation.
- **Aggregator:** Combines code and explanation into one answer (maybe formatting code nicely and then writing paragraphs).
- **Result to user:** The code followed by the explanation. Also display: "Experts used: Python model (cost 50 tokens), AWS model (cost 75 tokens)."

In a monolithic approach, one would send the entire original prompt to GPT-4, which would produce code and explanation in one shot (or in a few back-and-forth turns). DCAR instead achieved the same result via two targeted calls (using potentially smaller models) with user oversight.

5.4. Configurable Modes

DCAR supports multiple modes to adapt to user preference:

- **Manual Mode (Default):** Users explicitly select which domains or experts to use. This provides maximum control but requires user input on each query.
- **Auto Mode:** The system automatically selects experts based on rules or a learned policy. The user might trust the system and skip manual checks. The system could optionally summarize the choice ("Using Python and AWS experts for this query").
- **Budget Mode:** A user can set cost or latency budgets. E.g., "under $0.01 per query" or "respond under 2 seconds". DCAR then chooses the smallest combination of experts that can (likely) answer the query under that budget.
- **Hybrid Mode:** If the system is uncertain, it can ask a quick clarifying question ("This query may need the 'Database' model. Include it?") or provide a confidence slider.

The choice of mode would depend on the application. For exploratory tasks, Auto Mode is convenient; for budget-sensitive or security-sensitive tasks, Manual or Budget mode gives control.

5.5. System Implementation Considerations

Implementing DCAR requires a control layer that can interact with multiple models. This could be built as:

- **API Gateway:** A middleware service receiving user queries (e.g., via HTTP). It calls internal or external model endpoints. Many components (analyzer, router, aggregator) can be microservices.
- **Plugin for Chat UI:** In a chat interface (like VS Code Copilot or Slackbot), hooks can route messages through DCAR.
- **On-Prem vs Cloud:** Expert models could be locally hosted (for speed and privacy) or remote. DCAR can mix on-prem and cloud models transparently.

Regardless of deployment, the key is that DCAR sits *above* model APIs and orchestrates calls.

5.6. Relation to MoE

DCAR's routing is analogous to the Mixture-of-Experts concept, but at a higher level. In MoE, a trained gating network chooses expert sub-networks per token [10]. In DCAR, a separate routing system chooses entire models per query (or per query segment). One could view each expert model as a giant "neuron" in a MoE, with DCAR's policy engine as the gate g(x). The difference is that DCAR's decisions are exposed to and influenced by the user.

This has trade-offs. MoE inside a model automatically optimizes for accuracy (the gating learns which expert best handles each token). DCAR relies on classification heuristics and user input, which may be less precise but more controllable.

6. Experimental Methodology

We outline how to evaluate DCAR experimentally. Although building DCAR fully is beyond this paper, we describe a **simulation experiment** that a researcher or practitioner could implement to measure the benefits and costs.

6.1. Baseline and DCAR Scenarios

- **Baseline:** The standard approach: all queries are sent to a single large model (e.g., GPT-4). We assume an API cost (e.g., \$0.03 per 1K tokens) and record metrics: total input tokens, output tokens, latency, and model type.
- **DCAR:** The proposed routing approach: queries are first classified into domains, then each domain is handled by a specific model (which may be smaller/cheaper). We simulate both Manual and Auto modes: in Auto mode, the system routes according to a fixed rule set; in Manual mode, the ideal user choices are assumed known.

We can simulate DCAR by actually calling different models. For example, for a "Python coding" query, call an open-source code model (like CodeLlama or StarCoder) instead of GPT-4. For a "DevOps" query, call a Docker-specialized model (or a smaller general model with a "Docker context"). If some expert is not available, one could approximate by prompting the general model with a system message (though that muddies the cost comparison). The key is to have representative alternatives for each domain.

6.2. Dataset Construction

We need a dataset of realistic developer queries. One approach: - Collect 1000–5000 prompts from sources like StackOverflow, GitHub Copilot usage logs (if available), or synthetic generation covering domains: code, config (Docker, Kubernetes), cloud (AWS/GCP), data (SQL/NoSQL), general Q&A, math, etc. - Label or filter each prompt by domain tags. (This can be partly automated by keyword matching or a text classifier.) For instance, any prompt mentioning `"` function, `import python, or [.]py file` gets "Coding"; any prompt mentioning "Dockerfile" gets "DevOps"; mentions of "SELECT, JOIN" get "SQL/Database", etc. - Ensure diversity: some prompts may have multiple tags (e.g., code+cloud).

The dataset should mimic a developer's real workflow: mix of quick factual questions ("What's the syntax for this?"), code generation requests, explanation and documentation tasks, and multi-step tasks.

6.3. Models and Implementations

- **Analyzer:** We might implement a simple tagger: e.g., use a regex- or ML-based text classifier to assign 0-3 domain labels per prompt. We record its accuracy (in Auto mode, classification errors can hurt DCAR performance; in Manual mode, assume ideal tagging).
- **Expert Models:** For each domain tag, select a model:
- *Code:* e.g. StarCoder or Code Llama (13B).
- *DevOps/Config:* e.g. GPT-3.5 or a fine-tuned smaller model, if available. If no free model exists, we might approximate with GPT-3.5 with a hint prompt.
- *Cloud:* similar approach, possibly use GPT-3.5 or smaller.
- *SQL:* e.g. a 6B model tuned on SQL (if available) or GPT-3.5.
- *General English:* a small general model (e.g., Llama2 7B) for tasks like summarization.
- *Math:* e.g. GPT-4 if we consider math difficult, or use specialized tool like WolframAlpha (though external).
  Ideally, we use at least one open-weight model (like LLaMA2, Mistral, or open source variants of the GPT-3.5 family) for cost reasons.

Each model is prompted appropriately. For example, the code model might receive: `"<system> You are a helpful Python programming assistant. </system><user> ...original prompt... </user>"`. This ensures it stays on topic.

- **Fallback and Chaining:** For simplicity, our initial experiments might not implement iterative fallback (like escalating if confidence is low). We may assume either a fixed one-shot routing or a simple 2-layer cascade (cheap model then expensive model if needed). More complex chaining is future work.

6.4. Metrics to Collect

For each prompt in the test set, we will record: - **Token Counts:** Total input tokens and total output tokens for each model invoked. (Sum them for DCAR vs. baseline.) - **Latency:** Measured or estimated. We can measure actual API latency, or model inference time. For parallel calls (if any), use the maximum latency plus overhead. - **Cost:** Based on token counts and known pricing. If models have different pricing (e.g., CodeLlama free, GPT-4 costly), compute a synthetic dollar cost per query.
- **Accuracy/Quality:** Evaluate the correctness or quality of answers. This might require manual or automated scoring (e.g., programming prompts could be compiled/tested; explanations could be rated). Alternatively, use a held-out set of answers or "oracle" responses from GPT-4 for comparison. - **Expert**

Selection Accuracy: (For Auto mode) measure how often the Analyzer's domain tags match the ground truth tags. An error here could send a query to a wrong expert or miss an expert.

6.5. Experimental Conditions

We would run at least two conditions:

- **DCAR-Auto:** Analyzer tags automatically → Router selects experts automatically. (No user in loop.)
- **DCAR-Manual:** Analyzer tags and shows to user, assume user always selects exactly the needed experts (an upper bound on DCAR performance).
- **Baseline (Monolithic):** All queries go to GPT-4 (or another fixed large model).

Comparisons: We compare Baseline vs. DCAR-Auto vs. DCAR-Manual on total tokens, cost, and quality. Ideally, DCAR-Auto should approach the efficiency of DCAR-Manual, showing that automated routing is viable with good tagging.

6.6. Hypothetical Evaluation Pipeline

1. **Data Preparation:** Tag each query with the "true" needed expert set (perhaps by human annotation or by ground truth analysis).
2. **Run Baseline:** Send all queries to the large model, record output and tokens.
3. **Run DCAR-Auto:** Use the Analyzer to tag domains for each query, route to experts, collect outputs. Compute tokens.
4. **Run DCAR-Manual:** Use the true tags to route to experts (simulate perfect decisions), collect outputs.
5. **Aggregate Results:** Compute average token reduction, cost reduction (percentage and absolute), and average answer quality (e.g., pass rate, BLEU/ROUGE vs. baseline).

We note that DCAR may sometimes involve multiple models, so we should define how to compare output quality. In cases where DCAR uses multiple experts, one can evaluate each sub-answer separately or the combined answer as a whole. We might require that the combination is at least as good as baseline (for developer tasks, partial correctness might suffice).

6.7. Expected Outcomes

We anticipate that: - DCAR will show **significant token/cost savings**. For example, if 70% of queries only require one expert, and that expert is half the size of GPT-4, tokens (and cost) could drop ~50-70% for those queries. Even queries needing two experts would likely remain cheaper than one GPT-4 call if the experts are much smaller.
- **Latency:** There may be a trade-off. DCAR-Auto could parallelize experts (so total time ~ max(single model time)), or run sequentially (time = sum). If parallel, latency is roughly that of the slowest expert model. If sequential, some latency overhead exists. Careful scheduling (e.g., calling cheap model first, then heavy model only if needed) can mitigate this.
- **Quality:** We expect comparable or slightly lower quality. For pure factual or code tasks, a specialized model may even outperform a general model (as it is fine-tuned on code). The risk is if the Analyzer mis-tags (DCAR-Auto errors) or the Aggregator mis-merges. These factors must be quantified.

These experiments will validate whether DCAR meets its goals of efficiency without undue loss of capability.

7. Evaluation Metrics and Expected Results

In reporting results, we will use the following metrics:

- **Token Reduction (%):** 100·(1 – (DCAR_tokens / Baseline_tokens)).
- **Cost Reduction (%):** Similar formula using cost.
- **Average Latency:** Compare mean response time. Note if DCAR latency variance differs from baseline.
- **Answer Accuracy / Satisfaction:** Could be quantitative (e.g., a score) or qualitative (developer survey). We might define thresholds: "acceptable" vs "incomplete".

We hypothesize:

- *H1:* DCAR-Auto will reduce tokens by >50% on average across typical dev queries, and DCAR-Manual by even more (since it never errs).
- *H2:* Latency will remain in an acceptable range (<1 second for most queries) if expert models are not too slow; in the worst case, latency may increase modestly but within user tolerance (developers often wait a few seconds for results).
- *H3:* Task performance (code correctness, explanation accuracy) will not significantly decrease. In the best case, a Python-coding expert will output more idiomatic code than a general model. If DCAR-Auto mis-tags a query (false negative), performance may suffer; quantifying this error rate is key.

These hypotheses must be validated, but we expect DCAR to show a clear efficiency advantage for narrow tasks.

8. Discussion

The proposed DCAR/VEO approach involves trade-offs and raises questions:

- **Complexity Overhead:** DCAR is more complex than a single API call. It requires building and maintaining the Analyzer, routing logic, and managing multiple models. This is an engineering cost. However, for large-scale deployments or critical applications, the potential cost savings and control may justify this.

- **Context Sharing:** If a user query involves multiple experts, DCAR must ensure that all relevant context is available to each. For example, if a prompt has two questions ("A and B"), how do we feed context to each model? One approach is to duplicate shared context into each sub-prompt. Another is a more intelligent decomposition. This is an open challenge.

- **Output Consistency:** Different models may have different writing styles or verbosity. Aggregating their outputs could produce a disjointed answer. A solution is to have a final polishing step by a single model to harmonize style. Alternatively, the GUI can present each expert's answer in separate sections.

- **Failure Modes:** If an expert model fails (timeout or error), DCAR can fall back to a general model for that sub-task. The routing policy should handle such fallbacks gracefully. Logging and monitoring are important.

- **Security and Privacy:** Routing queries to specialized models could improve security if sensitive data is handled by local or in-house models. On the other hand, exposing routing policies might leak hints about the query content if not secured.

- **Cost Estimation:** The system should track costs of each expert model. Some experts may have per-token pricing, others fixed. DCAR can then pick cheaper experts when budgets are tight. This adds another dimension to routing decisions.

- **User Experience:** Presenting too many toggles or confirmations could annoy users. The interface must balance transparency with simplicity. Analytics could suggest default settings (e.g., "You use Default mode 80% of the time; switch to Auto?").

Despite these challenges, we believe DCAR's benefits (especially for high-volume or cost-sensitive environments) warrant this direction.

9. Incremental Adoption Strategy

DCAR can be adopted gradually in practice:

- **Plugin Integration:** Build DCAR as a plugin or extension for popular IDEs or chatbot frameworks. Initially, it could simply detect keywords and pop up suggestions ("Would you like to use a Python model for this?") without changing the pipeline.

- **API Gateway:** In enterprise settings, deploy DCAR as an API gateway. Applications continue to call an LLM API, but behind the scenes DCAR splits calls to specialized endpoints. This is analogous to how service meshes route HTTP requests.

- **Hybrid Mode:** Offer DCAR as an *opt-in* feature: for trusted or admin users, whereas others still use monolithic. This allows testing real-world workflows.

- **Open Ecosystem:** Encourage community development of expert models. If many organizations contribute open models (e.g., code LLMs, legal LLMs, etc.), DCAR can route to a rich catalog of experts.

- **Economic Incentives:** Show cost savings on bills to IT: e.g., "By using our Python expert, we saved \$500 in API costs last month." A dashboard could visualize token usage per domain.

In summary, DCAR does not require replacing existing models; it only adds a smart orchestration layer. Even partial deployment (some queries routed, others not) can yield benefits.

10. Limitations

It is important to acknowledge limitations:

- **Non-Expertise Overlap:** Some tasks do not neatly decompose. E.g., creative writing or abstract reasoning might span multiple domains unpredictably. DCAR's static tags may not fully capture such needs.
- **Model Availability:** For some domains, no smaller expert model may exist. The fallback is to use the large model, reducing the benefit. The effectiveness of DCAR depends on having good experts.
- **Routing Errors:** Automatic classification will sometimes misfire (false positives or negatives), leading to missing a needed expert or invoking an irrelevant one. This could slightly degrade output. Mitigating this (via confidence thresholds or user override) is crucial.
- **Additional Latency:** If multiple experts are invoked sequentially, DCAR could take longer than a single model (especially if networked APIs). Caching and parallel execution can help but not eliminate the issue. Thus DCAR may not suit ultra-low-latency requirements.
- **Evaluation Difficulty:** Measuring "answer quality" is nontrivial when the answer is a composite of multiple parts. Our evaluation plan may need customization for each use-case.

These limitations mean DCAR is not a universal solution, but rather a specialized tool for developer workflows where domains are well-defined.

11. Sustainability and Energy Considerations

One of DCAR's motivations is reducing energy use and carbon emissions of AI inference. By avoiding unnecessary computation in a huge model, DCAR should lower total energy consumption. For example, if a domain-specific model uses 10x less energy per inference than GPT-4, and it can handle many queries alone, the savings accumulate.

Furthermore, DCAR can integrate with energy-aware scheduling: expert models could be deployed on greener datacenters or spun down when idle. Cost models can include carbon pricing: e.g., prefer on-device inference to cloud if it has a lower carbon intensity.

Strubell *et al.* (2019) emphasize energy costs of training [6], and Faiz *et al.* (2024) model inference carbon [7]. DCAR adds to this conversation by targeting operational efficiency. In large deployments (e.g., an organization making millions of LLM calls per year), DCAR could cut energy use by >50%. This aligns with broader calls for sustainable AI practice (see [27] and [29]).

12. Conclusion

We have presented **DCAR with VEO**, a novel orchestration framework for AI inference. By routing developer queries to specialized models with explicit user control, DCAR aims to dramatically improve efficiency of foundation model usage. This approach leverages the fact that most practical queries are narrow in scope, and that many smaller models exist which can answer them cheaply. Unlike hidden internal optimizations, DCAR's visible policies empower developers to manage costs and trust the system.

This work shifts the focus of AI research from always scaling up model size to also improving *how* models are utilized. We encourage the community to explore infrastructure and middleware as first-class citizens in AI systems. Moving forward, we plan to implement a prototype DCAR platform and conduct user studies with developers.

Our hope is that DCAR-style architectures become part of the future AI stack: the "microservices of AI." Just as Kubernetes transformed software deployment, we envision DCAR and similar frameworks transforming AI deployment — making it modular, efficient, and user-centric.

13. Glossary

- **Artificial Intelligence (AI):** Systems that perform tasks which normally require human intelligence, often by learning from data [12].
- **Machine Learning (ML):** Subfield of AI where algorithms improve through experience or data [13].
- **Deep Learning (DL):** A type of ML using neural networks with many layers [2]. Commonly used for images, speech, and text.
- **Neural Network:** A computational model composed of interconnected units (neurons) with learnable weights.
- **Token:** A discrete piece of text (word, subword, or character) into which text is split [15]. NLP models process tokens rather than raw characters.
- **Tokenization:** The process of splitting text into tokens [15]. Methods include word-level, character-level, or subword (e.g. Byte-Pair Encoding).
- **Embedding:** A numeric vector representation of a token [18]. Similar words have similar embeddings.
- **Attention:** A mechanism that lets a model focus on different parts of input. In a Transformer, multi-head self-attention computes each token's representation by weighting all tokens [4].
- **Multi-head Attention:** Running multiple attention operations (heads) in parallel, each focusing on different aspects of the input [4].
- **Transformer:** A neural network architecture based on self-attention (no recurrence) [2]. Foundation for models like BERT and GPT.
- **Foundation Model:** A large model (usually Transformer-based) pretrained on broad data, adaptable to many tasks [1]. Examples: GPT-3, BERT, DALL·E.
- **Large Language Model (LLM):** A very large Transformer (billions of parameters) trained for language tasks [5].
- **Mixture-of-Experts (MoE):** An architecture where multiple sub-models (experts) exist, and a gating network routes inputs to a subset of experts [10]. This yields a sparse activation.
- **Prompt:** The input text provided to a model (often including instructions) to elicit a desired output.
- **Inference:** Running a trained model to get predictions or output (as opposed to training).
- **Aggregator:** In DCAR, the component that merges outputs from multiple experts into one final response.
- **DCAR (Developer-Centric Adaptive Routing):** Our proposed framework for routing queries to expert models.
- **VEO (Visible Expert Orchestration):** The policy layer in DCAR that is exposed to developers for selecting and reviewing model routes.
- **AI Gateway:** A middleware layer that manages calls to one or more AI models, handling tasks like routing, logging, and cost control (see [34]).

14. References

- Vaswani *et al.*, “Attention Is All You Need,” *NeurIPS 2017*, ArXiv:1706.03762 [3].
- Stanford HAI, “Brief Definitions of Key Terms in AI,” Apr. 2022. (Stanford HAI Explainer) [2] [1].
- Brown *et al.*, “Language Models are Few-Shot Learners,” *NeurIPS 2020*, ArXiv:2005.14165 (GPT-3).
- Radford *et al.*, “Improving Language Understanding by Generative Pre-Training,” OpenAI, 2018 (GPT-1).
- Strubell, Ganesh, McCallum, “Energy and Policy Considerations for Deep Learning in NLP,” *ACL 2019* [6].
- Faiz *et al.*, “LLMCarbon: Modeling the end-to-end Carbon Footprint of Large Language Models,” *ICLR 2024* [ArXiv:2309.14393] [7].
- Hoffman *et al.*, “Mixture-of-Experts in Large Language Models: A Survey,” *ArXiv 2023*.
- Touvron *et al.*, “LLaMA: Open and Efficient Foundation Language Models,” *ArXiv 2023*.
- Meier *et al.*, “DeepSeek R1: A 671B-parameter State-of-the-Art Open-Weight Model,” *X AI Blog, 2025*.
- Wolfe, “Mixture-of-Experts (MoE) LLMs,” Substack, 2023.
- Stone *et al.*, “Language Processors as Information-Working Systems: Word Embeddings,” *Journal of NLP*, 2016.
- Pinecone, “What is a Vector Database?,” May 2023 [20].
- OpenAI System Card (GPT-4), Apr. 2023.
- TrueFoundry, “Leading AI Gateway for LLM Workload Optimization in 2026,” blog (Jan 2026) [8] [9].
- Abdul Wahed, “AI Orchestration: The Microservices Approach to Large Language Models,” *Medium, Jan 2026* (conceptual).
- Radford *et al.*, “Language Models are Unsupervised Multitask Learners,” OpenAI, 2019 (GPT-2).
- Devlin *et al.*, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,” *NAACL 2019*.
- Raffel *et al.*, “Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer,” *JMLR 2020* (T5).

Note: All citations are linked to publicly accessible sources or model documentation.

---

[1] [2] [11] [12] [13] [14] [24] Brief Definitions of Key Terms in AI | Stanford HAI
https://hai.stanford.edu/policy/brief-definitions-of-key-terms-in-ai

[3] [5] [17] [21] [22] [23] Large language model - Wikipedia
https://en.wikipedia.org/wiki/Large_language_model

[4] [19] Attention Is All You Need - Wikipedia
https://en.wikipedia.org/wiki/Attention_Is_All_You_Need

[6] [1906.02243] Energy and Policy Considerations for Deep Learning in NLP
https://arxiv.org/abs/1906.02243

[7] [2309.14393] LLMCarbon: Modeling the end-to-end Carbon Footprint of Large Language Models
https://arxiv.org/abs/2309.14393

[8] [9] Leading AI Gateway for LLM Workload Optimization
https://www.truefoundry.com/blog/leading-ai-gateway-for-llm-workload-optimization

[10] aclanthology.org
https://aclanthology.org/2025.findings-naacl.251.pdf

15 16 Tokenization in NLP: How It Works, Challenges, and Use Cases | DataCamp
https://www.datacamp.com/blog/what-is-tokenization

18 Word embedding - Wikipedia
https://en.wikipedia.org/wiki/Word_embedding

20 What is a Vector Database & How Does it Work? Use Cases + Examples | Pinecone
https://www.pinecone.io/learn/vector-database/