International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 01 , January - 2026

From Push Email to Push AI: BlackBerry-Inspired Efficiency in Large Language Models

Aryan Gautam
Department of Electrical and Computer Engineering, UCSB

Abstract

Artificial intelligence systems, particularly large language models (LLMs), increasingly face efficiency and scalability challenges due to stateless inference paradigms. In current deployments, each user query is processed independently, requiring repeated context loading and token-by-token generation on cloud-based accelerators, leading to redundant computation, increased energy consumption, and higher latency. This paper draws inspiration from the evolution of mobile email systems in the 1990s, particularly BlackBerry's push-based architecture, which replaced inefficient polling with persistent connections that delivered updates only when new information arrived. We propose Sleep–Wake Inference (SWI), a push-style inference framework for LLM-based systems that maintains a lightweight persistent connection between client and server. Under SWI, the model remains in a low-activity "sleep" state during repeated or redundant interactions, reusing cached or lightweight responses, and transitions to a "wake" state only when genuinely novel input requires full inference. Responses are then pushed back to the client, avoiding unnecessary recomputation. SWI enables reductions in computational cost, network bandwidth usage, and client-side memory and battery consumption, while improving real-time responsiveness. Beyond efficiency gains, this architecture supports proactive, event-driven assistants capable of delivering relevant insights without explicit prompting. By reinterpreting push-based communication paradigms for modern AI inference, this work highlights a practical path toward more efficient and personalized LLM systems.

1 Introduction

As I said earlier, history repeats itself, and in the case of technology, I think it will again. In the 1990s and 2000s, BlackBerry reshaped mobile communication by introducing push email [19]. Unlike the polling method, where devices repeated the same question again and again — "Do I have new mail?" — BlackBerry's innovation relied on a persistent lightweight connection between device and server. Emails were delivered instantly in real time, saving bandwidth and battery of the device used by the client [10]. Before the breakthrough, emails basically worked the same way as in the real world. The POP3 (Post Office Protocol 3) just treated emails like digital letters, and once a message was downloaded on the desktop, it was deleted from the server [16] . While this was sufficient for single-device use, it had a huge flaw: syncing between devices was impossible, and emails could be lost. POP3 worked on periodic polling, which meant that it checked for new mail at fixed intervals. That meant real-time mail was not possible [11] .Figure 1 illustrates the evolution of email system architectures from polling-based protocols to push-based systems [14].

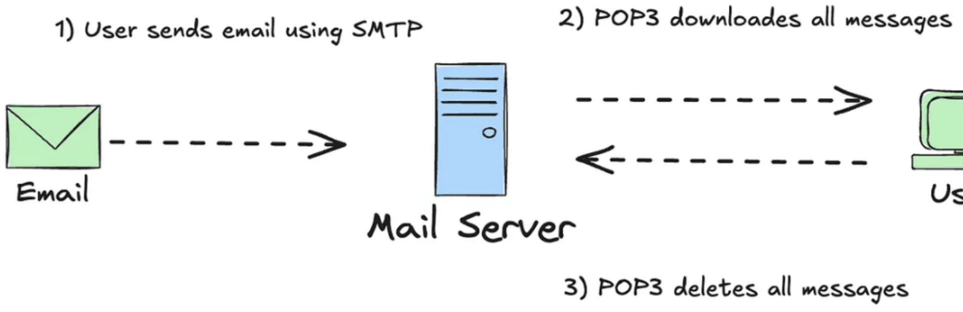
1) User sends email using SMTP
2) POP3 downloades all messages
Email
Mail Server
User
3) POP3 deletes all messages

Figure 1: Evolution of email system architectures. Reprinted from *The Evolution and System Design of Email Systems: From Postcards to Push Notifications* by Nikhil (2024).

The next improvement came with IMAP IDLE, which introduced a push-like mechanism. IMAP IDLE allowed servers to notify clients as soon as a new message arrived, enabling near real-time email delivery [4]. But it required maintaining an active connection with the server, draining the device's battery. For this reason, the model did not work well for mobile devices [11]. This was the problem that BlackBerry's push system solved. Rather than every device using polling, BlackBerry Internet Service handled the checks [19]. As soon as the server received a new email, it pushed the message instantly to the user's device. So, rather than asking the same question, "Did I get an email?", they built a main computer to do that for you. The benefits were that the model used data efficiently and also preserved battery life. This made real-time email practical on mobile devices. At its core, the push model is still fundamentally unchanged even decades later [10]. In my eyes, this progression would be similar for AI in the sense that just as early email protocols wasted resources with redundant checks, today's AI models waste resources by recomputing responses for every prompt, even when much of the context or query is repeated [2, 3, 5]. Recent work in prompt caching and attention reuse has started to address this inefficiency. Some examples include Prompt Cache [8], which introduced modular attention reuse for repeated prompt segments, while AttentionStore [7] and EPIC [9] explored context caching across conversations. In parallel, semantic caching approaches [18,24] use embeddings to detect duplicate or similar prompts, reducing redundant computations. Efforts such as Cached Transformers [23] and Recycled Attention [22] further improve long-context efficiency by reusing attention states. While these works are valuable, they remain fundamentally tied to the pull-based model of inference. This means that the client sends a request, and then the server always evaluates it. On the contrary, BlackBerry's push model shows a more efficient way of doing this [19]. This paper proposes Sleep–Wake Inference (SWI), which is a push-inspired approach to AI. Rather than recomputing for every prompt, the system maintains a lightweight connection and only "wakes up" the model for a new input. Redundant or semantically similar prompts are handled by cached responses [12, 21, 22]. Due to this small change, SWI has the potential to reduce compute costs, lower bandwidth usage, and also improve latency. This will be similar to the leap that BlackBerry made but for AI [10, 19].

2 Current State of AI Inference

Large Language Models (LLMs) such as GPT, LLaMA, and Claude have become the center of the AI revolution. Despite their impressive capabilities, their method of serving users remains rooted in a pull-based paradigm [1, 15, 20]. In most of the chatbots each client query is treated as a stateless request. This means that when a prompt is sent to the server, the model generates a response token by token from scratch. The system does not remember prior requests unless the full conversation history is explicitly recent. This was the main strength of BlackBerry's persistent push channel for email. Each interaction began and ended as an independent transaction [19].

The important question to ask is why every prompt is recomputed. The answer to this is because of the practical reasons. As LLMs are extremely large and have hundreds of billions of parameters and this must be hosted on clusters of GPUs or TPUs [17]. If we kept the model "always on" for every user it would be prohibitively expensive. Therefore, providers adopt a stateless design where resources are only allocated during active inference. But this causes redundancy. For example if two prompts share overlapping content the model still recomputes the same transformations every time. Sounds familiar to something we discussed earlier (polling). This was the same thing we were doing for early email protocols [10]. In simple terms, a pull-based system means that every time you ask AI a question, your device has to send the whole request to the server, and the server has to start from the beginning to generate an answer. Nothing is "saved" between the prompts unless the user manually resends the entire conversation history as part of the prompt. To give you a simple example, imagine you ask a chatbot ," What 's the weather in Santa Barbara today?" and then after that you ask "And what about tomorrow?". The system does not automatically know that the second question relates to the first. Due to this it reprocesses the entire context including the first question to kind of make sense of the second. By the example above you can see the problem of the system. The repeated prompts or overlapping context segments are processed from scratch. This means for each prompt the model uses full computation. The result is high GPU/TPU acceleration and that is why we say AI consumes so much energy as well as a lot of cloud cost [6, 17] . Due to this redundant approach or in technical terms a token by token approach users have to wait seconds before they get the answer.Energy is expended generating novel answers and for recomputing redundant parts of the context. In the real world these problems become even larger as cloud providers have to maintain massive infrastructure to handle millions of stateless requests daily. Mobile devices now have way better battery life than the 1990s but still repeated uploads or continuous polling consumes battery and data bandwidth unnecessarily. Due to the latency or delay of responses it makes interactions feel less natural just like the early emails.

3 Existing Approaches and Limitations

Before getting to my solution, I would love to shed light on some existing solutions. These solutions do reduce redundancy and manage long conversation contexts. The only thing that they lack is that they still treat every prompt as a stateless request. This section will look at the solutions, examine their limitations, and assess whether my solution could address these shortcomings. One of the early strategies focused on modular attention reuse. Prompt Cache [8] introduced a method in which intermediate attention states for repeated segments of prompts, such as system messages or standard instructions, are stored. By reusing these cached computations, the system avoids recalculating identical portions of input. Similarly, AttentionStore [7] went one step further and extended the idea to conversations, saving context across multiple turns. EPIC [9] further developed the idea and allowed cached content to be reused even if its position in the prompt changes. All of these methods did speed things up but still operated on a pull-based model, which meant that the client must send a request every time. Other works focus on detecting when two prompts are similar in meaning. Efficient Prompt Caching via Embedding Similarity [24] determines if a new prompt is close enough to a stored one and then reuses the old response. Adaptive Semantic Prompt Caching with VectorQ [18] improved this method by making the similarity test smarter. These approaches are close to what our SWI will achieve, but in this method, the server still has to evaluate each request before deciding whether to reuse an answer. A third group of methods also tries to handle long conversations. Cached Transformers [23] created memory caches inside the model, which means it does not need to process the entire history again. Recycled Attention [22] alternates between full and partial context processing. These methods saved time on long inputs but still followed the pull-based model. The main drawback of all these methods is that, since they function on the pull model, every request is separate, and the server never initiates action on its own. Each prompt triggers some form of computation. Users may also face issues as they might need to resend histories, which consumes significant bandwidth and battery.

4 Sleep–Wake Inference (SWI)

To understand how the Sleep–Wake Inference works, we first have to understand how BlackBerry's Push model worked. The main reason why this model was revolutionary was because it did not perform the repeated "checking" that we have mentioned so many times earlier in this paper. In this section, we are going to see in detail how this model actually worked. I have provided a diagram below to see this in detail.

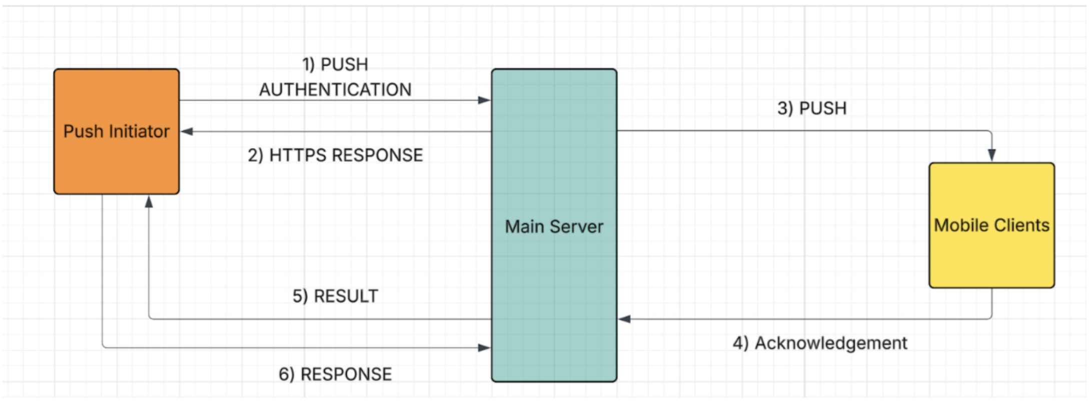

Figure 2: Sleep–Wake Inference (SWI) architecture showing the AI Gateway, cache, novelty check, and inference engine interaction.

4.1 BlackBerry Push Model as a Systems Blueprint

The process starts on the Push Initiator. In the real world, this could be an email provider (such as Gmail, Yahoo Mail, or a corporate Exchange server) or an enterprise system within a company. The work of the Push Initiator is to detect new content, typically an email, that now needs to be given to the user's device [10, 19].

In traditional POP3/IMAP systems, this role didn't exist in the same centralized way. Instead, the client (the user's device) was responsible for checking directly with the main server if new messages were available. BlackBerry shifted this work. So rather than the server being hit with thousands of devices with the same question, "Do I have mail?" the Push Initiator handled the job once and passed it to the BlackBerry network. This removed unnecessary duplication [17].

Push Model Operation. The BlackBerry push email system operates through a sequence of coordinated steps that ensure reliable and efficient message delivery.

1. **Authentication and Message Submission.** Before sending this content downstream, the Push Initiator establishes a connection with the Main Server. This step is called authentication, or proving identity. It ensured that only the correct email sources could deliver content to the user. At this point, the initiator also sent the new email message upstream to the Main Server.

2. **Server Acknowledgment.** The Main Server then sent back a confirmation to the Push Initiator that it had received the message. This was the equivalent of an HTTPS "OK."

3. **Message Push to Clients.** At this step, the Main Server became the central actor. It monitored incoming streams from Push Initiators and pushed messages directly to the Mobile Clients (BlackBerry handsets). This was different from POP3/IMAP, where the server was passive and waited for the client to ask for updates.

4. **Client Acknowledgment.** The Mobile Client then sent an acknowledgement back to the Main Server. This ensured reliability. If the acknowledgement failed, the server attempted retransmission. POP3/IMAP clients assumed the delivery was complete once the data was downloaded. In situations where the connection dropped, users might lose messages with no way to undo this. BlackBerry made acknowledgement a base feature. For this reason, it became the most reliable way to email. That is what made it the backbone of many government communications [17].

5. **Delivery Result Notification.** Once the acknowledgement was received, the Main Server sent a delivery result back to the Push Initiator. This confirmed whether or not the push to the device was successful.

6. **Loop Completion.** In some cases, the Push Initiator also sent a final confirmation back to the Main Server to complete the loop.

4.2 Why Push Fails in Email but Succeeds in BlackBerry

The model was better as it decreased latency. Users could receive email in real time. Second, it improved energy efficiency. In an era when battery life was limited, this change was significant as BlackBerry's model allowed devices to remain in a low-power listening state until the server had new content to deliver. Third, it optimized network bandwidth. By eliminating constant polling traffic, BlackBerry made it possible to run multiple phones on the data provider.

All these changes were good, but the main one was reliability. The model made sure to come out with some kind of an outcome. This confidence is what made BlackBerry the phone of that time. From a systems design perspective, the push model demonstrated the value of central orchestration. The Main Server acted as a mediator that coordinated requests, managed authentication, and handled delivery [10, 19].

4.3 Sleep–Wake Inference Architecture

The Sleep–Wake Inference (SWI) model is a proposed paradigm for improving the efficiency of large language model (LLM) inference by shifting from today's reactive, pull-based design to an event-driven persistent connection framework. Inspired by the push method discussed above, SWI has the same goals. It would reduce redundant computation and lower latency. This would also improve energy efficiency by allowing models to "sleep" when no novel input is present and "wake" only when meaningful new prompts arrive. In the context of SWI, the term "sleep" does not imply shutting down the system entirely. Instead, "sleep" refers to a low-power mode where the LLM remains in a low-compute state and does not actively engage in inference. In this state, GPUs or TPUs are not cycling to recompute tokens, nor are they reprocessing repeated context. Instead, the responsibility shifts to the Gateway, which quietly monitors input streams, caches previous results, and determines whether new computation is actually required. Essentially, the LLM "wakes" only when the Gateway identifies input that is truly novel or in simple words, new. So the main question is: how does the system know when a prompt is new? For this, I would be using research that has already been done. Adaptive Semantic Prompt Caching with VectorQ [18] has demonstrated methods for comparing incoming prompts to stored embeddings and using learned thresholds to decide whether two prompts are semantically equivalent. If a new query is within the similarity threshold, then a cache is requested, and the system can serve the prior response directly, bypassing the model.

4.4 Sleep–Wake Inference Execution Flow

The diagram below shows an AI Gateway that acts as the mediator between the client and a large inference cluster. The Gateway maintains a persistent session, runs a lightweight novelty check against a cache, and only wakes the heavy inference engine when the incoming prompt is truly novel. On the diagram, a decision diamond shows the route of the flow. Cache-hit responses are returned immediately, which means the model stays asleep. If the cache misses, then only the model is awakened. The model's normal state is sleep/idle, and it only becomes active on demand.

I will explain what happens in the same way I described the BlackBerry model.

1. **Client Initialization and Authentication.** The process starts from the Client (User App). In the real world, this could be a user-facing app such as a chatbot interface, a productivity tool embedding an LLM, or even a corporate workflow system requesting AI responses. The client authenticates with the AI Gateway Main Server.

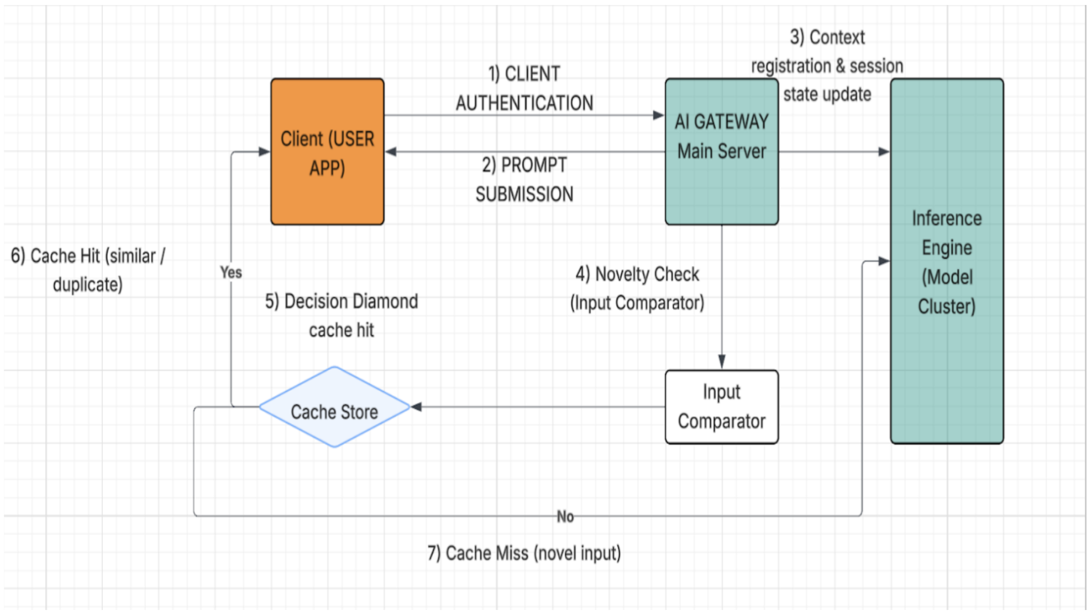
1) CLIENT AUTHENTICATION
2) PROMPT SUBMISSION
3) Context registration & session state update
Client (USER APP)
AI GATEWAY Main Server
Inference Engine (Model Cluster)
6) Cache Hit (similar / duplicate)
Yes
5) Decision Diamond cache hit
4) Novelty Check (Input Comparator)
Input Comparator
Cache Store
No
7) Cache Miss (novel input)

Figure 3: Sleep–Wake Inference (SWI) architecture showing the AI Gateway, cache, novelty check, and inference engine interaction.

2. **Prompt Submission.** Once authentication is done, the client sends the user's prompt to the AI Gateway. In pull-based systems, the client would have to send the entire conversation history every time, but here only the new piece of input (a delta) is transmitted.

3. **Context Registration and Session State Update.** The AI Gateway then performs Context Registration and Session State Update. It keeps track of the conversation, remembers history, and links new input fragments with cached knowledge. Unlike traditional LLM APIs where the server is stateless, here the Gateway actively manages session continuity.

4. **Novelty Check.** The Gateway then runs a Novelty Check using the Input Comparator. This mechanism checks whether the new input is truly novel or just a repeat (or a close variant) of a request that has already been answered.

5. **Decision via Cache Lookup.** The Novelty Check leads to the Decision Diamond. If the input matches something in the cache, the flow goes to the Cache Store.

6. **Cache Hit Path.** In the case of a Cache Hit (similar or duplicate), the cached response is returned directly to the client. The inference engine stays asleep, and the client gets the response in near real time.

7. **Cache Miss Path.** If the novelty check results in a Cache Miss (novel input), the Gateway pushes the query to the Inference Engine (Model Cluster). This "wakes up" the model, runs the necessary computation, and produces a fresh response, which is then passed back to the client and stored in the cache for future use.

Once the inference engine responds, the Gateway passes the output back to the client. To complete the cycle, the client sends an Acknowledgment (ACK) confirming receipt. This ensures reliability, as the Gateway knows the response was successfully delivered.

To put it all together, the Sleep–Wake Inference (SWI) model represents a shift from wasteful repetition to intelligent orchestration. In pull-based systems, the client acts as a historian, resending information and forcing the inference engine to recompute work it has already done. SWI eliminates this duplication. The AI Gateway acts as a persistent mediator, maintaining session state and filtering out redundant queries. It only wakes up the heavy inference engine when it is necessary [18]. The result is a system that is faster, leaner, and more reliable. Responses arrive in real time for cached inputs. GPUs and TPUs conserve power because they remain idle until novel work appears. For this reason, bandwidth use is reduced, as the system only sends incremental updates (deltas) when new data is seen. By keeping intelligence centralized in the Gateway, the system improves security and ensures consistency across millions of clients.

From a design perspective, SWI highlights the same architectural insight that made BlackBerry's push model revolutionary. It does not scatter the work across countless devices; instead, it concentrates it in one place. Similar to how BlackBerry servers checked once for new mail instead of letting every device check endlessly, the AI Gateway checks once for novelty and shields the model cluster from a flood of redundant prompts [13]. In short, the Gateway decides when to compute so that the inference engine can focus purely on what to compute. This division of responsibility is what makes SWI scalable and sustainable in the era of large AI models

5 Challenges and Barriers to Implementation

Discuss challenges in implementing SWI. Despite the clear potential benefits of a push-style, there are some reasons why the SWI has not been implemented yet. These challenges stem from many factors. These factors include practical challenges as well as technical challenges.

Large Language Models (LLMs) often have hundreds of billions of parameters and are hosted on GPU or TPU clusters. Maintaining a persistent, always-on connection with a million users poses challenges in resource allocation. Unlike email servers, inference engines perform heavy computation for every token generated. Keeping these engines in a semi-active or low-power state for each user could increase idle overhead and complexity in resource scheduling.

SWI relies on accurate caching and semantic comparison of prompts to determine whether the model needs to "wake." Implementing this requires very sophisticated similarity detection mechanisms. We also need robust embedding storage and consistent state management across sessions. While research on caching does exist, it is difficult to integrate into the current pull-based systems, let alone into a new push system [18].

Maintaining persistent, low-latency connections across mobile and web clients at global scale is technically very demanding. Other factors include network interruptions and firewall restrictions. User mobility can also make it challenging to guarantee reliable session continuity. Pull-based APIs avoid these complications by treating each request independently.

All the above problems could be solved if we really had the resources and money. Current cloud providers are heavily optimized for stateless inference because the model aligns with billing and scaling practices. The economics make sense, as it is easier to bill requests. SWI's persistent connection and caching may complicate cost allocation, making it harder to predict infrastructure needs and pricing.

The reason BlackBerry was successful is because the transition they made to the push system did not require changing the way things were done in the market, as they integrated the push mechanism into the existing system [13]. The software infrastructure for AI today that includes frameworks, APIs, and orchestration tools is largely built around the pull-based request–response system. Shifting to a push-driven system would require redesigning both client and server components, updating APIs, and retraining operators to manage a fundamentally different workflow. This is a major factor in why such a change would take significant time and effort, despite the potential efficiency gains.

Even though the security of the system is good, the core is still from the 1990s. In today's day and age, where cyberattacks have become much more common than they were three decades ago, the model is vulnerable. If session tokens are compromised, then continuous sessions would be vulnerable to attacks.

It is worth noting that push-based systems are not entirely foreign. Though they are not yet at the scale of SWI, push-based interactions have already started to emerge. On-device AI assistants like Siri, Google Assistant, and Alexa implement lightweight push-like behavior by listening for wake words and only activating heavy inference when triggered. In large-scale implementations, providers are experimenting with event-driven inference pipelines (e.g., stock market updates, monitoring dashboards, or IoT feeds), which automatically trigger model evaluations.

In future work, we plan to build and evaluate a prototype system based on the Sleep–Wake Inference (SWI) model. The goal is to develop a mini chatbot that demonstrates the feasibility of this approach. The planned evaluation would involve training the model on approximately 1,500 prompts, testing it on 300 prompts that are semantically similar to the training data, and then evaluating performance on an additional 200 completely novel prompts. Beyond this initial prototype, we aim to scale the system to larger workloads in order to study its behavior under more realistic usage conditions.

6 Conclusions

Technology is actually moving way faster than we expected. AI is here to stay, and for that reason, if we really want it to make a difference in our future, we need to make it better so we have a future. All the energy and resources that AI is using nowadays are causing a lot of harm to the environment, and something like SWI can reduce that. SWI addresses inefficiencies in pull-based inference through persistent connection, caching, and novelty detection, creating a faster and more reliable system. However, our study does have its limitations. Even though we were able to build a mini chatbot that highlights the practicality of this approach, the process still has gaps that need to be filled. SWI is not the whole framework but rather a step toward a more efficient and scalable AI system. Current cloud systems are optimized for stateless inference, and making the transition to a push system would require a complete overhaul. Despite all the challenges, I still feel the concept has potential, and given the right resources and mentorship, the model would be able to work in the correct direction.

References

[1] Anthropic. Claude: A next-generation large language model, 2024. Anthropic Technical Report.

[2] Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

[3] Mark Chen, Alec Radford, Rewon Child, Jeffrey Wu, Heewoo Jun, David Luan, and Ilya Sutskever. Efficient transformers for long contexts. *arXiv preprint arXiv:2301.08201*, 2023.

[4] Mark Crispin. Internet message access protocol – version 4rev1. RFC 3501, 2004. Defines IMAP and IDLE-based near real-time email delivery.

[5] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc Le, and Ruslan Salakhutdinov. Transformer-XL: Attentive language models beyond a fixed-length context. In *Proceedings of ACL*, 2019.

[6] Ricardo Fernandez, Daniel Lee, and Vinay Kumar. Energy consumption of large-scale ai models: Challenges and solutions. *Journal of AI Sustainability*, 2(1):15–27, 2025.

[7] Bo Gao et al. Attentionstore: Cost-effective attention reuse across multi-turn conversations. *arXiv preprint arXiv:2403.19708*, 2024.

[8] Ankit Gupta et al. Prompt cache: Modular attention reuse for low-latency inference. *arXiv preprint arXiv:2311.04934*, 2023.

[9] Tao Hu, Qi Zhang, and Lei Wu. Epic: Enhanced prompt indexing and caching for language models. *arXiv preprint arXiv:2402.05678*, 2024.

[10] Ken Krechmer. Real-time email delivery on mobile devices. *Communications of the ACM*, 45(12):43–49, 2002.

[11] Bernard Leiba. Imap4 idle command. RFC 2177, 1997. Introduces server push-style email notification.

[12] Xinyu Li, Yifan Wu, Yuchen Xu, and Min Chen. Cache-aware inference for large language models. *arXiv preprint arXiv:2310.01234*, 2023.

[13] Robert McQueen. Push vs. pull: Lessons from the mobile email revolution. *Tech Review*, 18(3):22–28, 2011.

[14] Nikhil. The evolution and system design of email systems: From postcards to push notifications. `https://blog.stackademic.com/the-evolution-and-system-design-of-email-systems-from-postcards-to-push-notifications-2024`.

[15] OpenAI. Gpt: Language models for general-purpose use, 2023. OpenAI Technical Report.

[16] Jon Postel. Post office protocol: Version 2. RFC 937, 1985. Early email retrieval protocol based on polling.

[17] Mohammad Samsi, Ricardo Fernandez, and Vinay Kumar. Cloud infrastructure for large-scale llm inference. *Journal of Cloud Computing*, 12(2):45–61, 2023.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 01 , January - 2026

[18] Luke G. Schroeder et al. Adaptive semantic prompt caching with vectorq. *arXiv preprint arXiv:2502.03771*, 2025.

[19] Glen Sweeny. *BlackBerry: The Inside Story of Research in Motion.* Wiley, 2009.

[20] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, and Armand Joulin. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[21] Shiqi Wu et al. Lightweight session management for persistent llm inference. *arXiv preprint arXiv:2406.07845*, 2024.

[22] Yuchen Xu et al. Recycled attention for efficient llm inference. *arXiv preprint arXiv:2404.05678*, 2024.

[23] Tianyi Zhang et al. Cached transformers: Memory caching for long sequence processing. *arXiv preprint arXiv:2312.01234*, 2023.

[24] Haoran Zhu, Bowen Zhu, and Jiantao Jiao. Efficient prompt caching via embedding similarity. *arXiv preprint arXiv:2402.01173*, 2024.