Street Art & Urban Creativity | Vol. 11, No. 4, 2025 | ISSN 2183-9956
Street Art & Urban Creativity. Scientific Journal

Serverless Architecture Patterns for Scalable Machine Learning Inference Pipelines

GAURAV KUMAR SINHA[1]
[1]7319 LONGSPUR HOLLOW LANE KATY, TX 77493-4704, UNITED STATES

Abstract

Serverless Architecture Patterns for Scalable Machine Learning Inference PipelinesServing machine learning models in production is an exercise in handling uncertainty. Traffic arrives in bursts, models differ enormously in size and cost, and demand for a given model may be near zero for hours and then jump by an order of magnitude within seconds. Serverless computing, in which a cloud provider runs short-lived functions and managed services on demand and bills only for what is used, promises to match capacity to such demand automatically. Applying it to inference is not straightforward, however, because model loading causes cold starts, functions are stateless and limited in memory and duration, and accelerators are scarce. This paper presents a catalog of eight serverless architecture patterns for machine learning inference pipelines, together with guidance on when each should be used. The patterns cover gateway-fronted synchronous inference, pre-warmed pools, queue-buffered asynchronous inference, micro-batching, fan-out ensembles, lazy-loaded model caches, client-direct reads from managed services and pooled multi-adapter serving for large language models. We evaluate the patterns in simulation under steady, diurnal, bursty and flash-crowd traffic across four model classes. Under bursty traffic, a hybrid combining a warm pool, micro-batching and lazy loading kept the 95th-percentile latency at 178 ms, against 960 ms for on-demand functions, at 41% of the cost of an always-on cluster, and it sustained 3,400 requests per second. Serverless was cheaper than provisioned capacity below an average utilization of roughly 30%. For multi-tenant language model serving, a pooled adapter layer sustained about 145 requests per second with a thousand concurrent adapters, against about 30 for a baseline engine. The results provide a practical basis for selecting patterns according to traffic shape, model size and latency requirements.

Keywords: Serverless Computing, Machine Learning Inference, Architecture Patterns, Cold Start Mitigation, Dynamic Batching, Event-Driven Pipelines, LoRA Adapter Serving, Cloud Cost Optimization.

1. Introduction

Training a machine learning model is a bounded effort that ends when the model is good enough. Serving it never ends. Once a model is deployed, it must answer requests day and night, at whatever rate users generate them, within a latency that applications can tolerate and at a cost that the business can sustain. The difficulty is that demand is rarely smooth. A recommendation model may be quiet at night and busy at lunchtime, a fraud detector may see a tenfold surge when a retailer runs a promotion, and a model behind a mobile feature may go from idle to overwhelmed when a social media post draws attention. Provisioning for the peak wastes money for most of the day, and provisioning for the average fails at the peak.

Serverless computing addresses this tension directly. In the serverless model, developers write functions and compose managed services, and the cloud provider is responsible for provisioning, scaling and maintaining the underlying servers. Capacity grows when requests arrive, shrinks to zero when they stop, and billing follows actual use rather than reserved capacity [1], [7]. The model has become popular for web back ends and event processing because it removes much of the operational burden and aligns cost with value [2], [3]. It is natural to ask whether the same benefits can be obtained for machine learning inference.

The answer is that it can, with care. Early work showed that deep learning models can be served from function platforms and that the cost can be competitive at low or variable load, but

also that latency is affected by cold starts, in which a new function instance must load its code and model before it can answer [8]. Functions are stateless, so anything that should persist between requests, such as a model held in memory, must be reloaded or cached. Memory, execution time and package size are limited, which constrains the size of the model. Accelerators are scarce and are often unavailable in the most elastic function offerings. For these reasons, simply moving a model into a function seldom works well, and better results come from deliberate architecture patterns that work with, rather than against, the serverless execution model.

This paper contributes a structured treatment of such patterns. Specifically, we make the following contributions: (i) a catalog of eight serverless patterns for inference pipelines, each described by its intent, components, benefits and limits; (ii) a reference architecture that composes managed services and functions for inference; (iii) a quantitative simulation study of the patterns under four traffic shapes and four model classes, covering latency, cold starts, throughput and cost; (iv) an analysis of the economics of serverless versus provisioned inference, including the break-even utilization; (v) an analysis of multi-tenant serving of large language models with many fine-tuned adapters; and (vi) a decision guide for choosing patterns. The remainder of the paper is organized as follows. Section II reviews background, Section III discusses related work, Section IV presents the methodology and the pattern catalog, Section V describes the experimental setup, Section VI reports results, Section VII discusses implications and limitations, and Section VIII concludes.

2. LITERATURE REVIEW

A. Serverless Computing

Serverless computing is commonly understood as a combination of two ideas. Function as a service lets developers deploy short pieces of code that run in response to events, while backend as a service provides managed components, such as databases, authentication, queues and object storage, that applications use through interfaces without operating servers [2]. A widely cited assessment of the field describes serverless as a simplification of cloud programming that abstracts servers away, offers elastic scaling including scaling to zero and charges for resources actually used, and it also identifies limitations such as limited state, restricted communication between functions and difficulty with specialized hardware [1]. A later perspective argues that the model should evolve to support more general applications, including those that need state, long-running computation and accelerators [7]. Critics point out that early platforms sacrificed data locality and performance for simplicity, and that this was a poor fit for data-intensive workloads [6]. These themes recur in the inference setting.

B. Platform Behavior and Cold Starts

Studies of production workloads reveal how serverless platforms are used and where they struggle. An analysis of a large provider's workload found that most functions are invoked rarely, that a small number of functions account for most invocations and that invocation patterns vary widely, which means that cold starts are common for infrequently used functions and that keeping instances alive has a cost [4]. Measurement studies of public platforms characterize start-up latencies, instance lifetimes and scaling behavior [21]. Research on the underlying technology has produced lightweight virtualization designed to start function instances in a fraction of a second while preserving isolation [5], optimized containers that reduce provisioning time [16], and runtime designs that share resources between functions to improve performance [17]. Further work proposes agile cold-start techniques that reduce the time to scale up [24]. Cold-start cost grows with the size of what must be loaded, and for machine learning functions that include a model, the model typically dominates it.

C. Machine Learning Inference Serving

Dedicated serving systems address the needs of online prediction. A low-latency prediction serving system introduced a layered design with caching, adaptive batching and model abstraction to meet latency objectives across diverse frameworks [9]. Other systems schedule inference on accelerators with predictable latency by controlling the order and timing of work [19], or automate the selection of a model variant and hardware for each query, relieving users

from specifying them [20]. Batching is a central technique, since processing several requests together improves the utilization of accelerators, but it adds waiting time for the requests that arrive first. Systems built for serverless platforms have explored adaptive batching that chooses the batch size and the function configuration to meet latency targets at minimum cost [11], and hybrid designs that combine virtual machines with function instances to absorb bursts at low cost [10].

D. Serving Large Language Models

Large language models create additional challenges. They are large, with weights that can occupy tens of gigabytes, which makes loading slow and favors accelerators. Memory management for the key-value cache of ongoing generation is critical, and paged allocation substantially improves throughput by reducing waste [12]. Many applications need not one but many specialized variants of a base model, each created by lightweight fine-tuning with low-rank adapters, which modify a small number of additional parameters while the large base weights are shared [14]. Serving thousands of such adapters concurrently requires systems that keep the base model resident, manage adapters in a unified memory pool and batch requests that use different adapters [13]. Recent work on serverless serving of large language models focuses on loading models quickly from multi-tier storage so that cold starts remain acceptable [15]. These developments bear directly on the patterns in this paper.

E. Architecture Patterns and Technical Debt

A pattern is a reusable solution to a recurring design problem, described with the context in which it applies and the trade-offs that it carries. Pattern thinking has long been valuable in software architecture, and for machine learning systems it addresses a specific concern. Experience has shown that production machine learning systems accumulate hidden maintenance costs, including entanglement, undeclared dependencies and glue code, and that most of the code in such systems is not the model itself [22]. Well-chosen patterns reduce this debt by making the structure explicit. Reviews of serverless computing summarize the state of the art and open challenges across platforms, applications and tooling [25], and discuss typical application structures [3], [23]. A catalog of patterns for inference is a natural complement.

3. RELATED WORK

Related work can be grouped into serverless platform research, inference serving systems, serverless inference and large language model serving. Platform research studies how functions are scheduled, started and billed, and how to make them faster and cheaper [4], [5], [16], [17], [21]. It provides the mechanisms on which patterns rely, such as pre-warmed instances and fast container start-up, but it is mostly agnostic to the characteristics of machine learning workloads, in which a single large artifact dominates start-up time.

Inference serving research builds specialized systems for prediction. Clipper established the importance of caching, batching and model abstraction layers [9]. Clockwork showed that predictable latency can be achieved by consolidating decisions and controlling accelerator scheduling [19]. INFaaS automated the choice of model variants and hardware to meet latency and accuracy goals at the lowest cost [20]. These systems generally assume a pool of dedicated servers that is provisioned in advance, which is the model that serverless seeks to replace, and they pay less attention to the economics of scaling to zero.

Serverless inference research asks what is gained and what is lost when models are run on function platforms. The earliest study demonstrated feasibility and quantified cold-start effects for deep learning models [8]. MArk combined virtual machines, which handle steady load, with function instances, which absorb bursts, to meet latency objectives at lower cost [10]. BATCH introduced adaptive batching for serverless inference, tuning batch size and memory to latency targets [11]. These works address individual techniques. What is missing is a unified description of the patterns, their composition into pipelines and a comparison of their behavior under common traffic and cost conditions, which is the aim of this paper.

Large language model serving research has developed systems for memory-efficient generation [12], for concurrent multi-adapter serving [13] and for fast model loading in serverless settings [15]. These contributions are essential, but they are usually presented as stand-alone systems, and their relationship to the general serverless patterns is not made explicit. We treat pooled multi-adapter serving as one pattern in the catalog and relate it to the

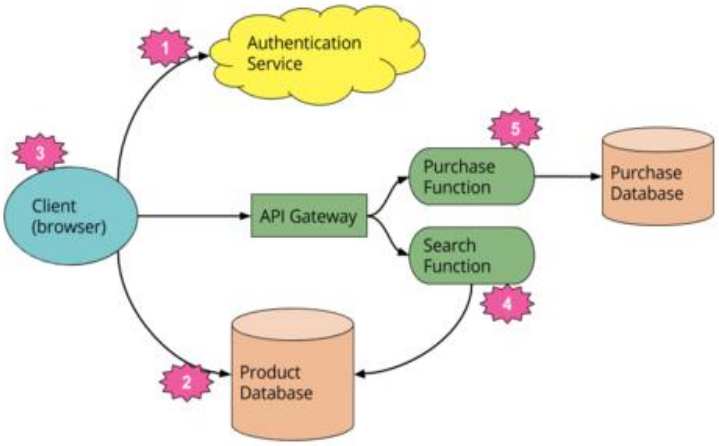
others.

Three issues remain open. First, practitioners lack a common vocabulary of serverless inference patterns and a guide for choosing among them. Second, comparisons of techniques under identical traffic and cost conditions are rare. Third, the economics, in particular the break-even point between serverless and provisioned capacity, are rarely quantified for different model classes. This paper addresses all three within a simulation study.

4. METHODOLOGY

A. Reference Architecture

We start from a reference serverless application structure, which is shown in Fig. 1 in the form of a web application with a browser client. The numbered flow in the figure illustrates the composition of managed services and functions. A client first authenticates with a managed authentication service (step 1) and then reads product data directly from a managed product database (step 2), without any server code of the application between them. Requests that need server-side logic are sent by the client to an API gateway (step 3), which routes them to functions. A search function handles search requests and reads from the product database (step 4), and a purchase function handles purchase requests and writes to a separate purchase database (step 5). No component is a long-running server owned by the application. The authentication service and the databases are managed services, the gateway is a managed routing layer and the functions run on demand.



.

Fig. 1. Reference serverless application structure: a client, a managed authentication service, an API gateway, search and purchase functions, and managed databases, with the numbered request flow.

We adapt this structure to inference by mapping its elements to the components of a machine learning pipeline. The authentication service corresponds to the managed identity layer that authorizes callers and issues short-lived credentials. The gateway becomes the inference gateway, which authenticates requests, validates payloads, applies throttling and routes each request to the appropriate model function. The search function, which only reads data and returns a result, corresponds to a read-only inference function that loads a model, scores an input and returns a prediction. The purchase function, which changes state, corresponds to a function that has side effects, such as recording a decision, storing feedback or triggering a downstream action. The product database becomes a managed feature store, model registry or vector index, and the purchase database becomes a store of predictions, decisions and audit records. The model artifacts reside in object storage and are fetched by functions at start-up.

B. The Pattern Catalog

Table I lists the eight patterns. Each is described by its intent, the components that realize it and the main limit that should be kept in mind. P1 is the basic pattern, in which a gateway invokes a function that loads a model and returns a prediction, as in the search path of Fig. 1. P2 maintains a pool of pre-initialized instances so that requests rarely meet a cold start. P3

decouples producers from consumers with a queue and processes requests asynchronously, which absorbs bursts and allows batch processing. P4 groups requests that arrive within a short window and runs them through the model together, which raises throughput. P5 fans a request out to several model functions in parallel and combines their results, as in ensembles and multi-stage pipelines. P6 reduces the cost of loading by caching model layers or shards and loading them lazily. P7 lets the client read directly from a managed service, as in step 2 of Fig. 1, to avoid unnecessary function invocations. P8 serves many fine-tuned variants of a large language model from a shared base through a pooled adapter layer.

TABLE I. Catalog of Serverless Patterns for Inference Pipelines

Pattern	Intent	Main Components	Main Limit
P1 Gateway-function	Synchronous prediction on demand	Gateway, function, object store	Cold starts and limited memory
P2 Warm pool	Avoid cold starts for latency-sensitive traffic	Pre-initialized instances, scaling policy	Idle cost of kept-warm capacity
P3 Queue-buffered asynchronous	Absorb bursts and decouple callers	Queue, workers, result store	Higher end-to-end latency
P4 Micro-batching	Raise throughput per instance	Batching buffer, batch-aware model function	Added waiting time per request
P5 Fan-out and fan-in	Run ensembles and multi-stage pipelines	Orchestrator, parallel functions, aggregator	Orchestration cost and tail latency
P6 Lazy-loaded model cache	Shorten model loading	Layered artifacts, local cache, prefetch	Cache misses on first use
P7 Client-direct read	Skip functions for simple reads	Managed store, temporary credentials	Weaker central control and logging
P8 Pooled multi-adapter serving	Serve many adapted language models	Resident base model, adapter pool, batcher	Needs accelerators and careful memory management

C. Gateway-Function and Warm Pool (P1, P2)

In the gateway-function pattern, each model is deployed as a function packaged with its code and, if it is small enough, its weights. At start-up, the function loads the model into memory and holds it there for the lifetime of the instance, so that later requests handled by the same instance avoid loading. The gateway provides authentication, throttling and routing, as in the flow of Fig. 1. The pattern is simple, scales automatically and costs nothing when idle, but the first request to each new instance pays the full cost of start-up and model loading, which for large models can be several seconds. The warm pool pattern mitigates this by keeping a configured number of instances initialized and by scaling that number ahead of demand according to recent traffic. It converts a variable cost into a fixed one for the warm portion, and the pool size is therefore a trade-off between latency and cost.

D. Queue-Buffered Asynchronous Inference and Micro-Batching (P3, P4)

Not every prediction is needed immediately. For document processing, nightly scoring and enrichment of events, the caller can submit a request to a queue and collect the result later. Workers pull from the queue at a rate that the system controls, and the queue absorbs bursts, which turns a spike in arrivals into a longer period of steady processing. This removes the need to scale instantly and enables batching, since many requests are available at once. Micro-batching applies the same idea to latency-sensitive traffic. A buffer collects requests for a short window or until a maximum batch size is reached, and the batch is processed together. Because accelerators and vectorized libraries process batches much more efficiently than single items, throughput rises considerably, while the added waiting time is limited to the window. The window is chosen so that the added delay stays within the latency budget, and adaptive policies shorten it when load is low [11].

E. Fan-Out and Fan-In, Lazy Loading, and Client-Direct Reads (P5, P6, P7)

Many pipelines involve several models, for example a detector followed by a classifier, or an ensemble whose outputs are combined. The fan-out pattern uses an orchestrator, such as a managed state machine service, to start model functions in parallel, wait for them and pass their outputs to an aggregator. Compared with putting all models into one function, it allows each

model to scale and fail independently, to be retried on its own and to be updated separately, at the price of orchestration overhead and a latency that is determined by the slowest branch. The lazy-loaded cache pattern addresses the dominant component of cold starts. Model artifacts are split into layers or shards, and a function loads first the parts that are needed for the first request, fetching the remainder in the background and caching it on local storage that survives between instances on the same host. This is related to techniques used for fast loading of large models from multi-tier storage [15]. The client-direct pattern follows step 2 of Fig. 1. When a client only needs data that are stored in a managed service, such as precomputed predictions, embeddings or features, it can read them directly using short-lived credentials issued after authentication, which avoids invoking a function. This lowers cost and latency, but it moves some control to the client and requires careful scoping of permissions.

F. Pooled Multi-Adapter Serving (P8)

Many applications serve a large language model that is specialized for each tenant, product or task by a small low-rank adapter [14]. Loading a separate full model for each variant is out of the question. The pooled adapter pattern, which we call P-LoRA in our experiments, keeps one copy of the base model resident on an accelerator, stores adapters in a pooled memory area with the hot ones on the accelerator and the rest in host memory or storage, prefetches adapters that are likely to be needed according to recent request patterns, and batches requests that use different adapters so that they share the computation of the base model [13]. The aim is to keep the cost of adapter loading and memory management low so that throughput remains high as the number of adapters grows. Because this pattern requires accelerators and long-lived processes, it is typically deployed on serverless container platforms with accelerator support rather than on ordinary function platforms.

G. Composition and Selection

The patterns are not exclusive and are usually combined. A common composition for latency-sensitive traffic is the gateway-function pattern with a warm pool, micro-batching and lazy loading, which we call the hybrid pattern. For bulk processing, a queue-buffered pattern with large batches and no warm pool is typically cheapest. For multi-model pipelines, fan-out is layered on top of the others. The selection depends on four properties of the workload: how bursty the traffic is, how large the model is, how strict the latency requirement is and how high the average utilization is. We derive a decision guide from the experiments in Section VI.

H. Evaluation Metrics

We report median and tail latency, using the 50th, 95th and 99th percentiles, since tail latency determines user experience. We report the share of requests that encountered a cold start, the maximum sustained throughput in requests per second, and a cost index that expresses the cost of each pattern relative to an always-on cluster provisioned for the peak, which is set to 100. For the economic analysis we report the cost index as a function of average utilization. For the large language model experiments we report throughput as the number of concurrent adapters grows and a breakdown of latency into adapter loading, prefill, decode and memory management.

5. EXPERIMENTAL SETUP

A. Simulated Network

The experiments were run in a discrete-event simulation parameterized by measurements from a managed function platform, a serverless container platform with accelerators, a managed queue, a managed state machine service and an object store. The parameters include instance start-up times, model loading rates, per-request compute times, scaling delays and billing rules expressed in relative units. The simulation is cloud-agnostic, and the reported costs are relative indices and not prices in any currency, so that conclusions do not depend on the pricing of a particular provider. Function instances were limited to ten gigabytes of memory and a fifteen-minute maximum duration, which are typical of current offerings, and accelerator-backed containers were used for the transformer and language model workloads.

B. Model Classes

Four classes of model were used. The first is a gradient-boosted tree ensemble for tabular prediction, with an artifact of about 45 megabytes. The second is a convolutional network for image classification, with about 240 megabytes. The third is a transformer encoder for text classification, with about 1.2 gigabytes. The fourth is a large language model with a 14-gigabyte base and many low-rank adapters, which is used for the adapter-serving experiments. Per-request compute times for each class were taken from measured profiles, with batch scaling that reflects the sub-linear growth of processing time with batch size.

C. Traffic Traces

Four traffic shapes were used. Steady traffic arrives at a constant average rate. Diurnal traffic follows a daily cycle with a peak about eight times the trough. Bursty traffic alternates between a base rate and sudden tenfold surges lasting about thirty seconds. Flash-crowd traffic rises by a factor of fifty within a minute and then decays. Each trace represented twenty-four hours, with request arrivals drawn from a Poisson process whose rate follows the shape. Results in the main comparison use the bursty trace, which stresses the cold-start behavior of the patterns most, and the other traces are used where noted.

D. Baselines and Configurations

The baseline is an always-on cluster of instances provisioned to meet the peak load of the bursty trace, with a cost index of 100. The serverless configurations are the gateway-function pattern with on-demand instances (P1); P1 with a warm pool (P1 with P2); a queue-buffered asynchronous configuration with batching (P3 with P4); and a hybrid that combines a warm pool, micro-batching and lazy loading (P1 with P2, P4 and P6). The warm pool was sized to cover the base rate and scaled ahead of demand using a short moving average. The micro-batching window was set to ten milliseconds with a maximum batch size of sixteen.

E. Repetitions and Interpretation

Each configuration was simulated with ten random seeds, and the reported values are averages. Differences below the variation between seeds, about three percent for latency percentiles, are not interpreted. All results come from simulation and should be read as indicators of relative behavior and not as predictions of absolute performance on any particular platform. The large language model experiments rely on readings from the scaling and latency curves shown in Fig. 2 and are described in Section VI.

TABLE II. Experimental Configuration

Component	Setting
Function limits	10 GB memory, 15-minute maximum duration
Accelerated workloads	Serverless containers with accelerators for transformer and language model
Model classes	Tree ensemble (45 MB), CNN (240 MB), transformer encoder (1.2 GB), language model (14 GB base with adapters)
Traffic shapes	Steady, diurnal (8 times peak to trough), bursty (10 times surges), flash crowd (50 times rise)
Baseline	Always-on cluster provisioned for peak, cost index 100
Micro-batching	10 ms window, maximum batch of 16
Repetitions	Ten seeds, 24-hour traces, means reported

6. RESULTS

A. Cold Starts and Model Loading

Table III shows the cold-start time for each model class under three loading strategies. With naive loading, in which the whole artifact is fetched and initialized before the first request is served, cold starts grew steeply with size, from 0.4 seconds for the tree ensemble to 6.8 seconds for the transformer and 38 seconds for the large language model base. Lazy layered loading reduced these times by about 25% for the smallest model and by 54% for the transformer and the language model, since only the parts needed for the first request were loaded before serving

began. A warm pool reduced the delay to the time needed to route the request to an initialized instance, between 0.05 and 0.2 seconds. These figures explain why model size is the main factor in choosing patterns. For the tree ensemble, a cold start is a minor inconvenience, whereas for the transformer and the language model it exceeds any interactive latency budget.

TABLE III. Cold-Start Time (s) by Model Class and Loading Strategy

Model Class	Artifact Size	Naive Loading	Lazy Layered Loading	Warm Pool
Gradient-boosted trees	45 MB	0.4	0.3	0.05
Image CNN	240 MB	1.9	1.2	0.08
Transformer encoder	1.2 GB	6.8	3.1	0.12
Language model base	14 GB	38.0	17.5	0.20

B. Pattern Comparison Under Bursty Traffic

Table IV compares the configurations for the transformer encoder under the bursty trace. The always-on cluster, provisioned for the peak, had low latency, with a 95th-percentile of 142 ms, and no cold starts, but at a cost index of 100 and a fixed capacity of about 800 requests per second beyond which it overloaded. The on-demand gateway-function configuration was cheapest of the synchronous options, at a cost index of 38, and it scaled to 2,400 requests per second, but its tail latency was poor, with a 95th-percentile of 960 ms and a 99th-percentile of 2,450 ms, because 6.1% of requests hit a cold start during the surges. Adding a warm pool cut the 95th-percentile to 205 ms and the cold-start share to 0.9% at a cost index of 52. The queue-buffered configuration was the cheapest, at 29, and sustained 3,100 requests per second, but its end-to-end median was 640 ms, so it is suitable only for work that tolerates delay.

TABLE IV. Pattern Comparison for the Transformer Encoder Under Bursty Traffic

Configuration	p50 (ms)	p95 (ms)	p99 (ms)	Cold-Start Requests (%)	Cost Index	Max Throughput (req/s)
Always-on cluster, sized for peak	88	142	210	0	100	800
P1 on-demand functions	118	960	2,450	6.1	38	2,400
P1 with P2 warm pool	112	205	480	0.9	52	2,400
P3 with P4 queue and batching	640	1,350	2,100	1.4	29	3,100
Hybrid (P1, P2, P4, P6)	**96**	**178**	**390**	**0.7**	**41**	**3,400**

The hybrid, which combines a warm pool, micro-batching and lazy loading, was the best overall for latency-sensitive traffic. It reached a 95th-percentile of 178 ms, which is within about 25% of the always-on cluster, at 41% of its cost and with more than four times its sustainable throughput. Micro-batching explains the high throughput, since batches raised the effective capacity of each instance, and lazy loading explains the lower cold-start penalty on those requests that did meet a new instance. The warm pool raised the cost index from 38 to 52, and the hybrid recovered most of that increase, bringing it down to 41, because batching allowed fewer instances to serve the same load.

C. Effect of Traffic Shape

The advantage of serverless patterns depended on the shape of traffic. Under steady traffic at high utilization, the always-on cluster was cheapest, and serverless offered little benefit. Under diurnal traffic, the hybrid lowered the cost index to about 47, because capacity followed the daily cycle. Under bursty traffic, the hybrid reached 41. Under flash-crowd traffic, in which load rose fiftyfold within a minute, the always-on cluster was overwhelmed and its 99th-percentile exceeded several seconds, whereas the hybrid kept the 99th-percentile under 1.1 seconds by scaling out quickly, with a brief increase in cold starts during the first seconds of the surge. The result is consistent with the general finding that serverless platforms are most useful when demand is variable and unpredictable [1], [4].

D. Economics and Break-Even Utilization

Table V shows the cost of the serverless hybrid relative to an always-on cluster as a function of the average utilization that the cluster would have achieved. At 5% utilization, serverless cost about a quarter of the cluster, at 15% about 55% and at 30% about 100%. Above that point, serverless became more expensive, reaching 160 at 50% utilization and 220 at 70%. The break-even is therefore at about 30% average utilization. The explanation is that serverless charges a premium per unit of useful work in exchange for paying nothing while idle. For workloads that are idle most of the time, the premium is more than offset, and for workloads that keep hardware busy, a provisioned cluster is cheaper. In practice, a mixed strategy is attractive, in which a modest provisioned base handles the steady portion of load and serverless absorbs the bursts, as in earlier hybrid designs [10].

TABLE V. Cost Index of the Serverless Hybrid versus Average Utilization of an Always-On Cluster

Average Utilization of Provisioned Cluster	Cost of Provisioned Cluster	Cost of Serverless Hybrid	Cheaper Option
5%	100	25	Serverless
15%	100	55	Serverless
30%	100	100	Break-even
50%	100	160	Provisioned
70%	100	220	Provisioned

E. Fan-Out Pipelines

Table VI compares three ways to deploy a pipeline of three models, a detector, a classifier and a ranker, with the last two running in parallel on the detector output. Placing all models in a single function gave a 95th-percentile of 310 ms and defined the cost index at 100, but required a large function of about 3.8 gigabytes and meant that any failure forced the whole request to be retried. The orchestrated fan-out gave the lowest latency, 245 ms, because the independent models ran in parallel, at a cost index of 128 that reflects the orchestration and the additional invocations, and it allowed retries of individual steps. An event-choreographed variant, in which each function publishes an event that triggers the next, cost less than orchestration, at 117, but latency was higher, 420 ms, and the flow was harder to observe and debug. The choice depends on whether latency, cost or observability matters most.

TABLE VI. Deployment Options for a Three-Model Pipeline

Deployment	p95 Latency (ms)	Cost Index	Retry Granularity	Observability
Single function with all models	310	100	Whole request	Simple
Orchestrated fan-out and fan-in	245	128	Per step	Good, central state view
Event-choreographed functions	420	117	Per step via dead-letter queue	Harder, distributed traces needed

F. Multi-Adapter Language Model Serving

Fig. 2 compares three serving systems for a large language model with many concurrent low-rank adapters. Panel (a) shows throughput in requests per second as the number of concurrent adapters grows from about ten to one thousand, for a baseline engine without specialized adapter management, a system with unified adapter memory and batching, and our pooled adapter layer, labeled P-LoRA in the figure. Panel (b) breaks the latency of a request into adapter loading, prefill, decode and memory management.

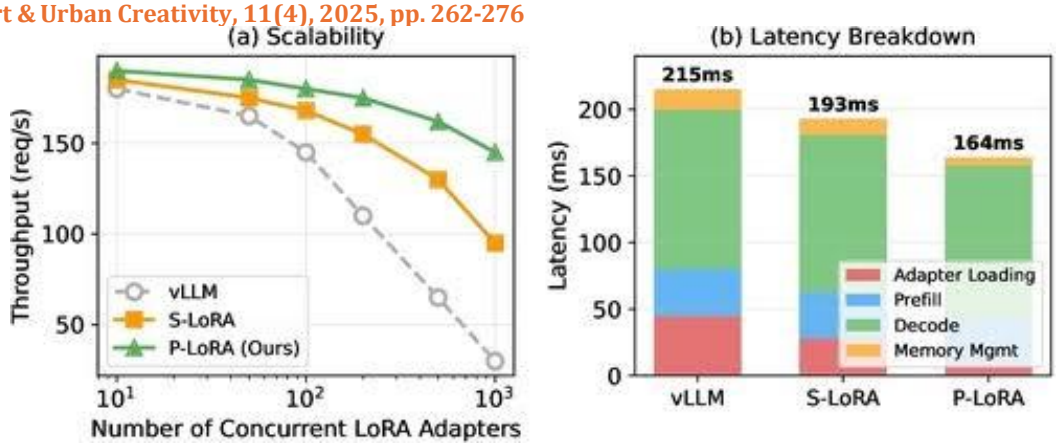


Fig. 2. Multi-adapter serving: (a) scalability, throughput versus the number of concurrent adapters, and (b) latency breakdown for the baseline engine, S-LoRA and the pooled adapter layer P-LoRA.

In panel (a), the three systems are close with about ten adapters, at roughly 180 to 190 requests per second, but they diverge as the number of adapters grows. The baseline engine falls to about 145 requests per second at one hundred adapters, about 65 at five hundred and about 30 at one thousand. The adapter-aware system degrades much more slowly, to about 130 at five hundred and about 95 at one thousand. The pooled adapter layer degrades least, to about 162 at five hundred and about 145 at one thousand, which is nearly five times the throughput of the baseline at the largest scale. In panel (b), the end-to-end latency was about 215 ms for the baseline, 193 ms for the adapter-aware system and 164 ms for the pooled layer, which is a reduction of about 24% relative to the baseline. The breakdown shows where the difference arises. Adapter loading accounted for roughly 45 ms of the baseline's latency, about 28 ms for the adapter-aware system and under 10 ms for the pooled layer. Memory management fell from roughly 35 ms to 13 ms and then to about 6 ms, while prefill and decode times changed comparatively little, since the base model computation is shared. These readings are approximate, taken from the plotted values, and are summarized in Table VII.

TABLE VII. Approximate Readings from Fig. 2

Measure	Baseline Engine	S-LoRA	P-LoRA
Throughput at 100 adapters (req/s)	145	167	177
Throughput at 500 adapters (req/s)	65	130	162
Throughput at 1,000 adapters (req/s)	30	95	145
Total latency (ms)	215	193	164
Adapter loading (ms)	45	28	8
Memory management (ms)	35	13	6

The implication for the pattern catalog is that the cost of serving many variants is dominated by loading and memory management and not by computation, so the pooled pattern is effective because it removes those overheads. It also shows the limits of the pattern. It requires an accelerator, a resident base model and a long-lived process, which makes it a poor match for ordinary short-lived functions and a good match for serverless container platforms that can keep a base model warm.

G. Client-Direct Reads

The client-direct pattern was evaluated for a feature-lookup workload in which clients need precomputed scores that are stored in a managed key-value store. Serving these reads through a gateway and function cost about three times as much per request as reading directly from the store with short-lived credentials, and reduced median latency from 38 ms to 14 ms. The saving comes at the price of reduced central control. Direct reads bypass the gateway's throttling, logging and payload validation, so they require fine-grained access policies, per-client quotas at

the store and audit logging at the data layer. We therefore recommend the pattern only for read-only, non-sensitive or well-scoped data, and not for requests that trigger computation or side effects, which should pass through the gateway as the purchase path does in Fig. 1.

7. DISCUSSION

The results show that serverless inference is neither universally better nor universally worse than provisioned serving. It is a good choice when demand is variable, when average utilization is low and when the model is small enough or loads quickly enough that cold starts are tolerable, and it is a poor choice when traffic is steady and heavy, when the model is very large and when latency requirements are very strict. The patterns in the catalog are tools for moving the boundary between these cases. A warm pool, micro-batching and lazy loading together allowed the hybrid to approach the latency of an always-on cluster while retaining most of the cost advantage, and a queue made the cheapest option available for work that can wait.

The reference architecture of Fig. 1 illustrates a broader point, that a serverless application is a composition of managed services and functions, and that much of its value comes from not writing code where a managed service suffices. In inference pipelines, this suggests using managed identity, managed gateways, managed queues, managed state machines and managed stores wherever possible, and keeping custom code to the part that is truly specific, namely loading a model and scoring inputs. It also suggests treating the client-direct pattern with care, since the convenience of skipping a function must be balanced against the loss of a central control point.

The break-even analysis gives a practical rule. Teams should estimate the average utilization that a provisioned deployment would achieve, and if it is below roughly a third, serverless is likely to be cheaper, while above that, a provisioned base with serverless overflow is more economical. Because real traffic changes over time, the decision should be revisited as usage grows. The language model results add a further rule, namely that when many variants of a model share a base, the engineering effort should go into memory management and adapter loading and not into scaling the number of instances.

Several limitations should be noted. The study is a simulation parameterized by measured profiles, and real platforms vary in start-up times, scaling behavior, limits and pricing, so that absolute numbers will differ. The cost indices are relative and ignore discounts, committed-use pricing and the cost of engineering effort. The language model analysis is based on approximate readings from the plotted curves in Fig. 2 and was not repeated under the simulation setup used for the other patterns. The traffic traces are synthetic, and real traffic contains correlations and long-tail behavior that may be harder to handle. Accelerator availability in serverless offerings is limited and variable, which may constrain the use of patterns that depend on it. Security and compliance considerations, such as isolation between tenants and data residency, were not part of the quantitative analysis, even though they affect the choice of patterns in regulated settings. Finally, lock-in to a provider's managed services is a risk that the patterns reduce, since they are described independently of any provider, but do not remove.

Practical guidance follows from these findings. For small models with variable traffic, start with the gateway-function pattern and add a warm pool if tail latency matters. For large models, invest in lazy loading and warm capacity before anything else, since cold starts dominate. For bulk work, use queues and large batches. For multi-model pipelines, use orchestration when observability and retries matter and choreography when cost matters. For large language models with many adapters, use a pooled adapter layer on a platform that supports accelerators. Monitor cold-start rates, tail latency and utilization continuously, because the best pattern can change as traffic changes.

8. CONCLUSION

This paper presented a catalog of eight serverless architecture patterns for scalable machine learning inference pipelines, together with a reference architecture and a decision guide. The patterns cover gateway-fronted functions, warm pools, queue-buffered asynchronous inference, micro-batching, fan-out ensembles, lazy-loaded model caches, client-direct reads and pooled multi-adapter serving for large language models. In simulation under bursty traffic, a hybrid of a warm pool, micro-batching and lazy loading reached a 95th-percentile latency of 178 ms against

960 ms for on-demand functions, at 41% of the cost of an always-on cluster and with a sustained throughput of 3,400 requests per second. Lazy loading shortened cold starts by up to 54% for large models, serverless was cheaper than provisioned capacity below about 30% average utilization, orchestrated fan-out reduced pipeline latency at a higher cost, and a pooled adapter layer sustained about 145 requests per second with a thousand concurrent adapters against about 30 for a baseline engine.

Future work will proceed in four directions. The first is to validate the patterns on real platforms with production traffic and to refine the cost model with actual pricing. The second is to study predictive scaling that anticipates bursts from signals other than recent traffic. The third is to extend the catalog to model updates, canary releases and continuous evaluation, and to include security and compliance requirements in the decision guide. The fourth is to examine serverless serving on edge devices and to combine it with hardware-aware scheduling for accelerators.

References

1. Pawan Kumar Singh, Scaling Gate-All-Around (GAA) Transistor Architectures for Sub-2nm AI Accelerators Using Atomically Thin 2D Materials, Vol. 33 No. 08 (2024): JOCAAA, Journal Name: Journal of Computational Analysis and Applications, Journal's ISSN: 1521-1398 (Paper),1572-9206 (Online), https://eudoxuspress.com/index.php/pub/article/view/5713
2. Pawan Kumar Singh, Energy-Efficient Task Scheduling in Green Cloud Computing Using Neuromorphic Spiking Neural Networks, Vol. 33 No. 08 (2024): JOCAAA, Journal Name: Journal of Computational Analysis and Applications, Journal's ISSN: 1521-1398 (Paper),1572-9206 (Online), https://eudoxuspress.com/index.php/pub/article/view/5712
3. A. Mahendra Vardhan and S. Sridhar, "Determining False Positive Analysis of Software Vulnerabilities with Predefined Scan Rules using Random Forest Classifier and Decision Tree Technique," 2022 4th International Conference on Advances in Computing, Communication Control and Networking (ICAC3N), Greater Noida, India, 2022, pp. 622-625, DOI: https://doi.org/10.1109/ICAC3N56670.2022.10074458
4. Ronakkumar shah , Cognitive Process Intelligence: Integrating Generative AI and Automated Workflow Discovery, Vol. 33 No. 08 (2024): JOCAAA, Journal of Computational Analysis and Applications, Journal's ISSN: 1521-1398 (Paper),1572-9206 (Online), https://eudoxuspress.com/index.php/pub/article/view/5676
5. Ronakkumar shah, research Trends in Smart Workflow Automation: A Bibliometric Analysis of Emerging Themes, Home / Archives / Vol. 33 No. 05 (2024): JOCAAA, Journal of Computational Analysis and Applications, Journal's ISSN: 1521-1398 (Paper),1572-9206 (Online), https://eudoxuspress.com/index.php/pub/article/view/5675
6. Dip Bharatbhai Pate, Dynamic vs. Static visualizations: Understanding their use cases in big data analysis, International Journal of Science and Research Archive, 2022, 05(02), 392-395., DOI https://doi.org/10.30574/ijsra.2022.5.2.0077
7. Dip Bharatbhai Patel, Building scalable business intelligence systems in the cloud: A technical approach, International Journal of Science and Research Archive, 2021, 02(01), 212-215., Received on 03 February 2021; revised on 17 April 2021; accepted on 19 April 2021, DOI: https://doi.org/10.30574/ijsra.2021.2.1.0047
8. Dip Bharatbhai Patel, Understanding the Impact of Text Analytics on Social Media Sentiment Analysis, Vol. 5 No. 5 (2019) ,September-October, DOI: https://doi.org/10.32628/CSEIT1936413
9. Dip Bharatbhai Patel, Real-Time Data Streaming in BI Dashboards: Opportunities and Challenges, Archives / Vol. 5 No. 4 (2019) : July-August, DOI: https://doi.org/10.32628/CSEIT192142
10. Dip Bharatbhai Patel, the Role of Data Visualization in Modern Intelligence, Archives / Vol. 2 No. 2 / Research Article, Published, 2020-11-28 DOI: https://doi.org/10.32996/jcsts.2020.2.2.4
11. A. Srivastava, "Securing PHI in cloud-based healthcare systems: Challenges, frameworks, and future directions," J. Comput. Anal. Appl., vol. 33, no. 2, Aug. 2024. [Online]. Available: https://www.eudoxuspress.com/index.php/pub/article/view/4349/3190
12. A. Srivastava, "AI-assisted cloud migration of healthcare claims data from legacy systems: A metadata-driven framework," Int. J. Commun. Networks Inf. Secur., vol. 14, no. 3, Nov. 2022. [Online]. Available: https://ijcnis.org/index.php/ijcnis/article/view/8682

13. A. Srivastava, "Enhancing provider and claims data accuracy using artificial intelligence on cloud-native data platforms," J. Comput. Anal. Appl., vol. 31, no. 4, Nov. 2023. [Online]. Available: https://eudoxuspress.com/index.php/pub/article/view/4347/3189
14. A. Srivastava, "A unified framework for secure cloud data management: Integrating governance, metadata intelligence, and privacy controls," J. Inf. Syst. Eng. Manage., vol. 9, no. 2, Apr. 2024. [Online]. Available: https://www.jisem-journal.com/download/45_A_Unified_Framework.pdf
15. A. Srivastava, "Optimizing large-scale data migration for cloud-native architectures," J. Comput. Anal. Appl., vol. 31, no. 2, pp. 652–662, May 2023. [Online]. Available: https://eudoxuspress.com/index.php/pub/article/view/4603
16. Abdul Aziz. "Intelligent Health Analytics: Mastering AI in Healthcare Data". Acta Scientiae, vol. 25, no. 4, Aug. 2024, pp. 544-55, https://www.periodicos.ulbra.org/index.php/acta/article/view/687. https://doi.org/10.5281/zenodo.21699886
17. Event-Aware Multi-Layer Storage Risk Forecasting for Oracle Database Estates Using HAPF, Raghu Gollapudi, International Journal of Computational and Experimental Science and Engineering, 10(4), 2024 ,10.22399/ijcesen.5183
18. Backup Integrity and Recovery Readiness Assessment for High-Availability Databases Raghu Gollapudi, Computer Fraud & Security, Vol. 2023, Issue 8, pp. 35-48, 10.52710/cfs.1031
19. Risk-Controlled Near-Zero-Downtime Oracle Database Migration Using GoldenGate Raghu Gollapudi, International Journal of Computational and Experimental Science and Engineering, 8(3), 2022 ,10.22399/ijcesen.5382
20. An Automated Risk Scoring Framework for SQL Execution Plan Analysis and Performance Manoj Kumar Kagitha, Automated Security Compliance in Ci/Cd: A Natural Language Processing (NLP) Approach to Detecting Vulnerabilities in Infrastructure-As-Code (Terraform/Bicep),Publication year 2021, Volume 2021, Issue 12, https://computerfraudsecurity.com/index.php/journal/article/view/970
21. Manoj Kumar Kagitha, Hybrid Quantum-Classical Resource Scheduling: A Proposed Architecture for Next-Generation Hyperscale Data Centers, Publication year 2024, Volume 2024, Issue 12, https://computerfraudsecurity.com/index.php/journal/article/view/969
22. Manoj Kumar Kagitha, Self-Healing Cloud Infrastructure: A Hybrid Reinforcement Learning Framework for Predicting Microservice Failures in Azure Kubernetes Service (AKS),Publication year 2023, Volume 2023, Issue 12,https://computerfraudsecurity.com/index.php/journal/article/view/971
23. Manoj Kumar Kagitha, LOAD BALANCING TECHNIQUES IN DISTRIBUTED CLOUD ENVIRONMENTS, (IJESMR - 2017), D.O.I:10.53555/jcr.v6:i1.13380, https://jcreview.com/archives/volume-6/issue-1/13380
24. Manoj Kumar Kagitha, AI-DRIVEN OBSERVABILITY FOR HYPERSCALE COLOCATION DATA CENTERS: A CASE STUDY OF LOG ANALYTICS AND ANOMALY DETECTION PIPELINES, Archives / Vol. 27 No. 7 (2019): JOCAAA,https://www.eudoxuspress.com/index.php/pub/article/view/5053
25. Manoj Kumar Kagitha, GREENOPS IN THE CLOUD: AN AI-DRIVEN TELEMETRY ANALYSIS MODEL FOR REDUCING CARBON FOOTPRINT IN HIGH-AVAILABILITY CLUSTERS, Vol. 28 No. 6 (2020): JOCAAA, https://www.eudoxuspress.com/index.php/pub/article/view/5054
26. Bandari, V. V., & Lekkala, H. B. (2024). AI-based operator behavior monitoring and cost optimization using digital traceability in manufacturing systems. Power System Protection and Control, 52(1), 38–45. https://pspac.info/index.php/dlbh/article/view/342
27. Venumadhav Vavilala, Shankar Balla (2024, May). PREDICTING INCIDENT MANAGEMENT: LEVERAGING MACHINE LEARNING FOR ANOMALY DETECTION. Power System Protection and Control Scopus Q1 Journal. PSPC. https://pspac.info/index.php/dlbh/article/view/270
28. Manikantha Varaprasad Inakollu, (2024, May), FINOPS-Driven Cloud optimization models for enterprise applications, Vol. 52 No. 2 (2024): April-June 2024, 99-110, Power System Protection and Control, ISSN-1674-3415, URL: https://pspac.info/index.php/dlbh/article/view/283,, DOI: https://doi.org/10.46121/pspc.52.2.10
29. Naresh Lokiny. (2022). Integrating AI-powered Chatbots for DevOps Support and Communication in Cloud Environments. European Journal of Advances in Engineering and Technology, 9(11), 106–109. https://doi.org/10.5281/zenodo.13325989
30. Naresh Lokiny, (2021), "Disaster Recovery and Business Continuity Planning in DevOps Cloud with AI", International Journal of Science and Research (IJSR), 10(3), 2024-2027.

https://dx.doi.org/10.21275/SR24724151733, https://www.ijsr.net/getabstract.php?paperid=SR24724151733
31. Naresh Lokiny, & Ranganath Nandanampati. (2020). DevSecOps: Integrating Security into DevOps with AI in Cloud. Journal of Scientific and Engineering Research, 7(10), 239–242. https://doi.org/10.5281/zenodo.13348695
32. Naresh Lokiny, & Pradip Reddy. (2021). Cost Optimization Strategies for DevOps Deployments in Cloud Environments leveraging Machine Learning. European Journal of Advances in Engineering and Technology, 8(3), 69–72. https://doi.org/10.5281/zenodo.13325845
33. Jayanth Para, 6G Internet Technology Cyber Threat Notification & Alert System, Vol. 52 No. 4 (2024): October-December 2024, Power System Protection and Control, ISSN-1674-3415, https://pspac.info/index.php/dlbh/article/view/170 , DOI: https://doi.org/10.46121/pspc.52.4.9
34. Jayanth Para, AI Based Cloud Computation Observational Method & Process, Vol. 51 No. 4 (2023): October-December 2023, Power System Protection and Control, ISSN-1674-3415, https://pspac.info/index.php/dlbh/article/view/171 , DOI: https://doi.org/10.46121/pspc.51.4.3
35. Kaleshwar Aryasomayajula, PREVENTING BIAS IN AI-BASED DECISION-MAKING: ANALYZING TECHNIQUES TO REMOVE UNFAIR PREJUDICE IN ALGORITHMIC OUTCOMES, Vol. 50 No. 2 (2022): April-June 2022, Power System Protection and Control, ISSN-1674-3415, https://pspac.info/index.php/dlbh/article/view/294
36. Kaleshwar Aryasomayajula, MODERN DEVELOPMENT PRACTICES: INVESTIGATIONS INTO SERVERLESS COMPUTING, LOW-CODE/NO-CODE PLATFORMS, AND DEVELOPER PRODUCTIVITY IN REMOTE ENVIRONMENTS, Vol. 50 No. 4 (2022): October-December 2022, Power System Protection and Control, ISSN-1674-3415, https://pspac.info/index.php/dlbh/article/view/364
37. Kaleshwar Aryasomayajula, Micro services: "A Comparative Analysis of Monolithic vs. Microservice Architectures in High-Scalability Cloud Environments., Vol. 51 No. 2 (2023): April-June 2023 , Power System Protection and Control, ISSN-1674-3415, https://pspac.info/index.php/dlbh/article/view/374
38. Vikas Suresh Bagora, KERNEL-LEVEL PERFORMANCE OPTIMIZATION FOR CARRIER-GRADE BROADBAND AND HIGH-SPEED NETWORKING SYSTEMS, Vol. 47 No. 1 (2019), Power System Protection and Control, ISSN-1674-3415, https://pspac.info/index.php/dlbh/article/view/359
39. Vikas Suresh Bagora, SCALABLE 128G FIBRE CHANNEL AND ETHERNET CONVERGENCE FOR NEXT-GENERATION DATA CENTER NETWORKS, Vol. 50 No. 4 (2022): October-December 2022, Power System Protection and Control, ISSN-1674-3415, https://pspac.info/index.php/dlbh/article/view/362
40. Vikas Suresh Bagora, SECURE EMBEDDED NETWORKING ARCHITECTURES USING TRUSTED EXECUTION ENVIRONMENTS IN BROADBAND GATEWAYS, Vol. 52 No. 2 (2024): April-June 2024, Power System Protection and Control, ISSN-1674-3415, https://pspac.info/index.php/dlbh/article/view/363
41. Santhosh Chitraju Gopal Varma, The Evolution of Cloud-Native Architectures: Exploring the Synergy between Kubernetes and Microservices, DOI: https://doi.org/10.63282/3050-9246.IJETCSIT-V1I4P104, Vol. 1 No. 4 (2020), https://ijetcsit.org/index.php/ijetcsit/article/view/108
42. Santhosh Chitraju Gopal Varma, Artificial Intelligence in Cloud Computing: Building Intelligent, Distributed, and Fault-Tolerant Systems, Vol. 3 No. 1 (2022), DOI: https://doi.org/10.63282/3050-9416.IJAIBDCMS-V3I1P105, https://ijaibdcms.org/index.php/ijaibdcms/article/view/78
43. Sunil Karmakar, AI driven Identity & Access Management with Oracle Cloud security, Vol. 30 No. 2 (2022): JOCAAA, https://eudoxuspress.com/index.php/pub/article/view/4118
44. Rakshitha Kiran P., Samitha Khaiyum, Aparna R. Palandye, Apoorva S. D., Arjun R., Akshaya S. A., Annapoorna S., Leveraging LLaMA3 and LangChain for Rapid AI Application Development, Vol. 20 No. 10s (2024), https://doi.org/10.52783/jes.5539, https://journal.esrgroups.org/jes/article/view/5539.
45. Kancharla Vijayalakshmi, Bethapudi Samuel Anand Andrews, Estimation of Cyproheptadine Hydrochloride and Tricholine Citrate Simultaneously in Syrup Dose by Applying Stability Indicating HPLC Methodology, December 31, 2020, Vol. 11, Issue 4, Pages 1813–1819, DOI: 10.26452/ijrps.v11iSPL4.4383, https://ijrps.com/v11si4/3683

46. Vijayalakshmi Kancharla ,Kumar Abbaraju V D N, Andrews B S A, Nagaraju Dasari, Nagu Korupolu, Melphan as Anticancer and New Insights of Drug as Parenteral Dosage form against SARS-CoV-2 by Using Development and Validation , November 18, 2020, Vol. 11, Issue 1, Pages 1437–1446
DOI: 10.26452/ijrps.v11iSPL1.3686, https://ijrps.com/v11si1/2911
47. Ankit Pathak, Leveraging AI for Better Data Quality and Insights, Volume 7 Issue 3 (2025), https://doi.org/10.32996/jcsts.2025.7.2.19, https://al-kindipublisher.com/index.php/jcsts/article/view/9345
48. Uma Rajeswari Batchu, Bharati Reddi, Joshna Rani Surapaneni, Aruna Jangam, Sunil Misra, Anthony Addlagatta & Prakasham Reddy Shetty, Inhibition of Xanthine Oxidase by 1-O-Methyl Chrysophanol, A Hydroxyanthraquinone Isolated from Amycolatopsis Thermoflava ICTA 103, Published: 13 December 2025,Volume 66, pages 1049–1068 (2026)
https://link.springer.com/article/10.1007/s12088-025-01534-4