Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Mitigating Cold Start Latency in Serverless Computing through LLM-Driven Optimization

Author: Rohan Murali Nair (2855534)

1st supervisor:	Daniele Bonetta
daily supervisor:	Matthijs Jansen
2nd reader:	Alexandru Losup

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

August 26, 2025

"I am the master of my fate, I am the captain of my soul"
from Invictus*, by William Ernest Henley*

Abstract

Serverless computing has become a popular cloud model because it covers much of the infrastructure management and allows developers to concentrate on the application logic. However, a major drawback is its limited use in latency sensitive applications due to the problem of *cold start latency*. Cold starts occur when a function is invoked after a period of inactivity, forcing the platform to provision resources, initialize containers and load code before the request is served. These delays, often lasting several seconds, hurt the user experience and make serverless platforms less suitable for time-critical applications.

Several techniques have been proposed to reduce startup time, including pre-warming, snapshotting and lazy loading. Although helpful, they usually come with trade-offs in efficiency, portability or adaptability. This thesis explores a different direction by combining large language models (LLMs) with structured inputs and observability to adaptively optimize cold start performance.

The framework breaks cold-start latency into seven stages using a profiler and turns the measurements into structured prompts. Based on these prompts, LLMs generate configuration-only patches for Knative service manifests. A closed-loop controller applies patches with built-in guarding policy and rollback policies, ensuring that changes remain safe and stable.

Experiments on three representative workloads (lightweight, dependency-heavy and network-bound) show cold start reductions of up to 65–84% compared to static baselines. These results indicate that structured, LLM-driven optimization provides a reliable and general approach to reducing cold-start latency in serverless systems.

Contents

List of Figures **iii**

1 Introduction **1**

1.1 Plagiarism Declaration 2

1.2 Thesis Structure 2

2 Motivation **3**

2.1 Research Questions 5

2.2 Research Methodologies 6

2.2.1 (M1) Design and abstraction 6

2.2.2 (M2) Experimental evaluation across LLM models 6

2.2.3 (M3) Comparative analysis based on workload 6

2.2.4 (M4) Extensibility through alternative optimization techniques . . . 7

3 Problem Formulation **8**

4 Design **10**

4.1 System Components 11

4.1.1 Continuum 11

4.1.2 Observability Stack 12

4.1.2.1 Stage Profiler 12

4.1.3 Prompt Generation 14

4.1.3.1 Inputs 15

4.1.3.2 How the prompt is generated 15

4.1.3.3 Why token budgeting 16

4.1.3.4 Example output from the prompt generator 17

4.1.3.5 Why this template matters. 17

4.1.4 Autonomous Optimizer 18

4.1.4.1 Scope of changes . . . 18
4.1.4.2 Plugin-based model interface . . . 18
4.1.4.3 End-to-end flow . . . 18

5 Evaluation **20**
5.1 Evaluation Setup . . . 21
5.1.1 Cluster VMs (Experiment Setup) . . . 21
5.1.2 LLM backends . . . 22
5.1.3 Procedure . . . 22
5.2 Continuous optimization over time . . . 22
5.2.1 Lightweight Functions . . . 23
5.2.2 Dependency-heavy functions. . . . 25
5.2.3 Network Bound Function . . . 27
5.2.4 Evaluation of the feedback loop and rollback policy . . . 29

6 Future Direction **31**
6.1 Machine learning model for optimization . . . 32
6.2 Beyond Knative configuration . . . 35
6.3 Better rollback policy . . . 35
6.4 Energy aware cold start tuning . . . 36

7 Conclusion **37**

References **39**

List of Figures

4.1 Framework architecture. 11
4.2 Classification of cold start latency into 7 Stages. 13

5.1 Baseline configuration on lightweight workload. 24
5.2 LLM-driven framework on lightweight workload. 24
5.3 Baseline on dependency-heavy workload. 26
5.4 LLM-driven framework on dependency-heavy workload. 26
5.5 Baseline Configuration on network bound workload. 28
5.6 LLM-driven framework on network bound workload. 28
5.7 Rollback policy on dependency-heavy workload 29

6.1 Framework architecture with a Multi-Armed Bandit Algorithm. 32
6.2 MAB-based tuning on lightweight workload (cold-start stages) 33
6.3 MAB-based tuning on dependency-heavy workload (cold-start stages) 34
6.4 MAB-based tuning on network bound workload (cold-start stages) 34

LIST OF FIGURES

1

Introduction

Serverless computing is one of the key innovations in modern cloud technology. Serverless computing helps developers focus on the application side, while infrastructure management is handled by providers. In 2014, Amazon introduced AWS Lambda, the first major serverless service that made the idea popular (1) and very quickly Microsoft Azure and Google Cloud followed by introducing their own Function-as-a-Service (FaaS) platforms. As serverless computing has improved over time, they became the most important part in the modern cloud computing era(2).

Serverless reduces the effort required to manage infrastructure, allowing new applications to be built and deployed quickly. The pay-per-use model makes serverless even more attractive, as it saves costs by removing the need to keep idle servers running (3).

However, the benefits of serverless computing come with its own limitations. A major challenge is vendor lock-in, as applications often become closely tied to a provider's platform. In addition, observability, debugging and compliance requirements can be difficult in managed environments. Among these challenges, **cold start latency** is one of the most important problems to adopt serverless applications for latency sensitive applications. Cold starts occur when a function is invoked after a period of inactivity, requiring the platform to allocate resources, initialize runtime and load application code before responding back to the request. These delays are often measured in hundreds of milliseconds or even seconds, can affect the user experience in applications where the faster response from the application is essential (4).

Over the years, researchers and engineers have proposed a variety of techniques to mitigate cold-start latency. Common approaches include container pre-warming, snapshotting execution environments, optimizing runtime initialization and predictive scheduling of resources. While each of these solutions shows really good progress, they often have trade-offs

related to resource consumption, workload-specific tuning or system complexity.

This thesis presents a framework that reduces cold start latency in Knative by using a stage profiler and a closed-loop optimizer that proposes configuration-only patches with an LLM. The profiler decomposes each cold start into seven stages and points out the current bottleneck while the system is running. The controller reviews each proposal against schema rules and tolerance bands, runs a quick test on live traffic, rolls back automatically if needed, and applies the accepted changes as new Knative revisions. The application code and images remained unchanged. The framework is evaluated on three workload types and three LLM models,compared with static baselines defined separately for each workload, each using a single fixed configuration that remains unchanged over time. This work demonstrates that combining stage-level observability, safe automation and an LLM driven optimizer at the Knative level provides a practical solution that consistently optimizes cold-start latency in serverless services.

1.1 Plagiarism Declaration

I confirm that this thesis work is my own work, is not copied from any other source (person, Internet, or machine), and has not been submitted elsewhere for assessment.

1.2 Thesis Structure

This thesis is structured as follows. **Chapter 2** presents the motivation and introduces the research questions. **Chapter 3** formalizes the problem, including objectives, assumptions, and evaluation metrics. **Chapter 4**4 describes the system design and key architectural decisions. **Chapter 5** details the evaluation on the various workloads, comparing both baselines and LLM optimized results. **Chapter 6** discusses future directions and potential extensions. **Chapter 7** concludes with a summary of findings and contributions.

2

Motivation

Serverless computing has grown into a widely adopted cloud model, yet its use is still shaped by performance challenges that appear during function startup. Among these, cold start latency stands out because it directly impacts user experience and limits the suitability of serverless platforms for latency critical applications such as interactive services, financial transactions and real time analytics (4, 5). Addressing this challenge is the main motivation for the approach proposed in this thesis, which seeks to reduce cold start delay during normal operation without changing application code or images.

To support this motivation, several techniques have been developed to reduce startup latency. For example, container pre-warming strategies maintain pools of pre-initialized containers that are ready to serve requests, thereby reducing cold-start delays. Lin and Glikson (6) demonstrate that pool based pre warming can reduce P99 latency by as much as 85%, though at the expense of resource overhead. Snapshot and memory prefetching were the approaches that were used in REAP (7), which accelerates the start by persisting execution states and preloading memory pages, achieving speeds of up to 3.7×. Another common method is the use of warm-up strategies, where platforms such as AWS Lambda rely on scheduled invocations or background activity to keep functions alive, trading improved performance for higher operational costs (8). Application-level optimizations, such as lazy loading of dependencies, defer initialization until required, thereby reducing initial latency but shifting the overhead to the first set of requests (4). Some research focuses on ML-specific workloads, where techniques such as model quantization and code trimming reduce memory footprints and initialization time, thereby lowering cold start latency, although often at the cost of reduced model fidelity (5). Finally, systems like WarmSwap and Pagurus (9) explore dependency sharing and reuse of inter-function containers, enabling millisecond-level cold starts but often requiring sophisticated platform support. Together,

these methods show a wide range of solutions but also recurring trade-offs in cost, portability, and generalizability.
In summary, existing solutions reflect three recurring themes:

- The performance improvements often come at a cost, such as higher resource usage or reduced accuracy.
- Many techniques remain platform specific, limiting portability across serverless ecosystems.
- Most strategies are static, lacking the adaptability needed for diverse and variable workloads.

Even with these improvements there are still important gaps. Keeping functions warm reduces delay but consumes alot of resources when there is no traffic. Lazy loading shifts the work to the first requests, so early users experience delays. This is a problem for real-time applications. Making code or models smaller to start faster can affect the quality of the results. Saving and restoring snapshots or sharing pieces between services often requires deep changes and adds day-to-day complexity. Most importantly, many methods use fixed rules and do not adjust as traffic or workloads change. In shared cloud environments, where many different services run together and demand can spike or drop without warning, these static approaches often fail.

To overcome these limitations, recent studies have been explored machine learning and more recently large language models (LLMs), for automated configuration tuning and optimization. Even though these solutions do not directly target optimization in serverless functions, they show that LLMs can be applied to optimize configuration parameters within feedback-driven systems. For example, GPTuner (10) leverages LLM to read database manuals and synthesize tuning configurations, while λ-Tune (11) demonstrates how LLMs can generate entire configuration scripts based on workload context,improving robustness compared to earlier approaches Similarly, LLM based approaches like SlsDetector (12) reveal the ability of models such as GPT-4 to detect and explain YAML based serverless configurations, with accuracy surpassing data driven methods. Studies such as LLMTune (13) and OtterTune (14) show the broader promise of LLM-guided configuration optimization, from database knob tuning to automated parameter selection in DBMSs. These systems show that LLMs can handle configuration reasoning and build end-to-end pipelines.
The promise of LLM driven optimization in this domain can be summarized in three points.

- Using pre-trained knowledge to reason over YAML manifests and system parameters.

- Dynamically generating targeted configuration patches without manual intervention.
- Enabling adaptive, workload-aware optimization that bridges the gap between static techniques and evolving runtime conditions.

Building on these ideas, this thesis explores how LLM-guided optimization can help reduce cold-start delays in serverless systems. A profiler breaks the latency into seven stages and produces signals for each stage, which are then passed to an LLM. Based on these inputs, the LLM proposes small YAML manifest changes to tune Knative service parameters. These changes are applied and tested in repeated cycles, creating a feedback loop that adapts as workloads shift. Rather than relying on fixed, static settings, the approach offers a more flexible and cost-aware way to cut cold-start latency by making use of the LLM's ability to reason over configuration. In this way, it connects existing static techniques with the need for runtime, workload-aware optimization.

2.1 Research Questions

To formalize the scope and contributions of this work, the following research questions are posed.

- **RQ1:** How can we design a flexible framework that supports different LLM models for automated performance tuning?
- **RQ2:** How do different LLM models influence the quality and performance outcomes of the optimization in the proposed framework?
- **RQ3:** How does the performance of LLM-based optimization vary across various workloads?
- **RQ4:** How can the framework be extended to include other optimization techniques in the same proposed solution?

Together, these research questions frame the contribution of this thesis by demonstrating how LLM-guided configuration optimization can complement and extend existing cold-start mitigation strategies.

2.2 Research Methodologies

2.2.1 (M1) Design and abstraction

To address RQ1 ("How can a flexible framework be designed that supports different LLM models for automated performance tuning?"), a pluggable optimization framework was developed for Knative services, with the possibility to extend it to other serverless platforms. The design followed an optimization loop with four main parts: (i) a deployment layer, powered by Continum (15)(ii) a stage-level profiler that records performance metrics, (iii) a prompt generator that translates these metrics into schema-checked instructions and (iv) an optimizer that proposes configuration changes. These components were placed inside a feedback loop, guided by a clear policy that decides when to accept, reject, or roll back changes. This design ensures adaptability to different LLM backends while keeping parameter tuning safe and consistent.

2.2.2 (M2) Experimental evaluation across LLM models

To address RQ2 ("How do different LLM models influence the quality and performance outcomes of the optimization in the proposed framework?"), a set of controlled experiments was carried out. The framework was connected to multiple LLM backends (ChatGPT, Claude, Gemini), all using the same prompt schema and acceptance rules. For each backend, repeated cold start events were triggered and stage-level metrics were collected. The evaluation measured the effectiveness of the optimizations, including latency reduction and stability of improvements, and then compared the influence of different models. The results indicate whether the choice of LLM backend has a noticeable effect on tuning effectiveness.

2.2.3 (M3) Comparative analysis based on workload

To address RQ3 ("How does the performance of the LLM-optimized framework vary across different workloads?"), the evaluation was extended to three representative workload types:

1. Lightweight functions
2. Dependency-heavy services
3. Network-bound applications

Each workload was deployed as a Knative service and tested under scale-from-zero events, both with a static baseline and with the LLM-driven optimization loop. Cold-start latency was decomposed into seven stages and comparisons were made between workload types.

This methodology shows how optimization strategies generalize across different services and whether stage-level bottlenecks change depending on the workload.

2.2.4 (M4) Extensibility through alternative optimization techniques

To address RQ4 ("How can the framework be extended to include other optimization techniques in the same proposed solution?"), the LLM-based optimizer was replaced with a non-LLM alternative, namely a Multi-Armed Bandit (MAB) algorithm. The feedback loop, acceptance rules, and rollback mechanisms remained the same to ensure a fair comparison. The experiments measured not only performance improvements but also how quickly the system adapted, its flexibility, and its stability under changing workloads. This evaluation demonstrates that the framework is not limited to LLMs. Its modular design supports different optimizers, confirming its flexibility as a general solution.

3

Problem Formulation

The primary challenge addressed in this thesis is the mitigation of cold start latency in serverless platforms. Cold starts occur when a function is invoked after being idle and no pre-initialized execution environment is available. In such cases, the platform must allocate resources, load the function image, initialize the runtime, and complete warm-up steps before the request can be served. These preparations introduce noticeable delays, commonly referred to as cold-start latency (4, 5).

Formally, let the cold start latency L be decomposed into the following stages:

$$L = L_{\text{sched}} + L_{\text{pull}} + L_{\text{init}} + L_{\text{runtime}} + L_{\text{proxy}} + L_{\text{app}} + L_{\text{req}},$$

Here, L_{sched} denotes the scheduling delay, L_{pull} the time taken to pull the container image, L_{init} the container initialization time, L_{runtime} the startup delay of the runtime, L_{proxy} the proxy warm-up cost, L_{app} the application-level initialization, and L_{req} the time required to handle the first request. Breaking down the cold start in this way aligns with empirical studies of Knative, which show that these components contribute differently depending on the workload and the underlying infrastructure (16). The aim of this thesis is to reduce the total latency L across a range of workloads and environments.

Serverless platforms such as Knative expose a set of tunable configuration parameters, denoted by $\theta = \{\theta_1, \theta_2, \ldots, \theta_n\}$, which influence these stages. Examples include `minScale`, `initialScale`, `concurrency`, and container resource limits (17). Selecting optimal values for θ is non-trivial because:

- The relationship between θ and L is highly non-linear and workload-dependent.
- Static heuristics often fail under dynamic, multi-tenant conditions (4).

- Exploring the parameter space exhaustively is infeasible due to time and cost constraints (7).

We therefore formulate the optimization problem as

$$\theta^* = \arg\min_{\theta \in \Theta} L(\theta, W, H),$$

where W represents the *workload type* (for example dependency heavy, DB like, or lightweight) and H denotes the underlying hardware and runtime environment. The goal is to find the optimal configuration θ^* which reduces the cold start latency while satisfying to correctness and cost efficiency constraints

$$\text{s.t.} \quad C(\theta) \leq C_{\max}, \qquad R(\theta) \leq R_{\max},$$

where $C(\theta)$ is the resource cost induced by configuration θ and $R(\theta)$ bounds resource usage such as memory or CPU.

To address this problem, this thesis investigates how large language models (LLMs) can be integrated into the optimization process. A profiler breaks down cold-start latency into its individual stages, and the LLM uses these observations to propose targeted configuration updates θ'. By applying updates iteratively and evaluating their effects, the system moves toward a more optimized configuration. θ^* (10) (11) (12).

In summary, the problem can be stated as:

> *Given a serverless function f, its workload W, and environment H, determine the optimal configuration parameters θ^* that minimize cold start latency L, subject to cost and resource constraints, using LLM-guided optimization.*

4

Design

The framework is built as a feedback-driven system that combines detailed cold-start profiling, observability, and an LLM-based optimizer into a single architecture. Its overall workflow is shown in Figure 4.1. In broad terms, the system monitors Knative deployments, breaks cold-start latency into separate stages and uses large language models to propose targeted configuration changes. These changes are then applied back to the deployment, creating a closed optimization loop inspired by traditional self-adaptive models such as MAPE-K (18).

To enable automated and repeatable deployment of cloud-native serverless applications, the framework relies on **Continuum** (15). Continuum simplifies infrastructure provisioning and orchestration on virtual machines, providing the base for deploying Knative services and managing their execution environments. This ensures that profiling and optimization experiments can be carried out in a consistent and reproducible way.

The observability layer consist of **OpenTelemetry** (19), which enables fine-grained tracing of Knative's request lifecycle. The collected metrics are visualized using **Jaeger UI** (20), providing structured insights into cold start events. In addition to OpenTelemetry, the framework integrates directly with the **Kubernetes API** to capture additional metrics that tracing alone does not expose. At the core of this layer lies the **Stage Profiler**, which leverages these combined sources to decompose cold start latency into multiple phases, producing fine-grained measurements that allow for precise identification of bottlenecks. These traces are then stored as a structured logs representing stage-specific latencies, which act as input to the autonomous optimizer.

The **Prompt Generation** module enhances the profiler output by adding contextual information, such as Knative YAML manifests and node specifications. This information

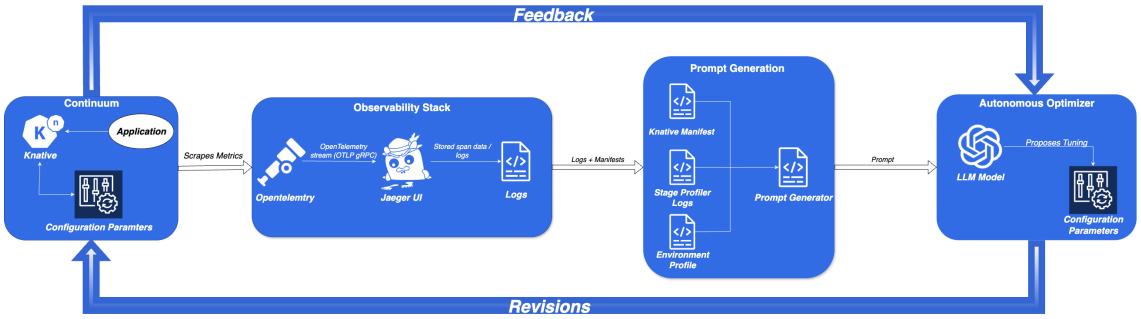

Figure 4.1: Framework architecture.

is transformed into structured prompts that highlight stage-specific latencies and configuration parameters. By framing system observations in a form that large language models can process, this component ensures the optimizer can reason about both performance bottlenecks and configuration constraints.

Finally, the **Autonomous Optimizer** integrates large language models into the feedback pipeline. Unlike static approaches, this optimizer is implemented as a flexible plug-in system that can host different LLM backends, including OpenAI's GPT, Claude or Gemini. The optimizer consumes stage-level profiling data and generates targeted Knative configuration patches (e.g., tuning `minScale`, `concurrency` or container limits). These patches are iteratively applied and re-evaluated, enabling adaptive, workload-aware tuning that achieve better tuning than fixed static methods

In general, the design follows the principles of self-adaptive systems, bringing together infrastructure automation, observability and LLM-driven optimization in a single architecture that directly addresses the cold-start challenge in serverless computing.

4.1 System Components

The framework is organized into four main components: (i) the deployment layer, (ii) the observability stack, (iii) the prompt generation module and (iv) the autonomous optimizer. Each part has its own role in the feedback-driven loop, and together they help reduce cold-start latency.

4.1.1 Continuum

Continuum (21) is a framework that automates infrastructure provisioning and application deployment across virtualized environments in the compute continuum. It provides

a structured workflow to instantiate virtual machines, configure required system dependencies and orchestrate cloud-native services with minimal manual intervention, making it easier to reproduce and repeat setups.

Continuum is employed as the foundational deployment layer. It provisions the underlying infrastructure virtual machines and automates the installation and configuration of the Knative platform alongside the target serverless application functions.By handling low-level infrastructure concerns, Continuum enables the creation of consistent, repeatable environments in which stage-aware profiling and LLM-driven optimization can be executed. This integration allows the framework to focus on measurement and tuning, while Continuum handles VM creation, platform setup, and service management in a dependable way.

4.1.2 Observability Stack

The observability layer uses distributed tracing and control-plane monitoring to fully record cold starts. It emits request-path spans using **OpenTelemetry** (19) and visualizes them in **Jaeger UI** (20), while also querying the **Kubernetes API** (22) for the official pod and container state transitions (e.g., scheduling and readiness). At the center of this layer is the *Stage Profiler*, which combines these data sources and generates both human-friendly traces in Jaeger and structured logs for use by the optimization components.

4.1.2.1 Stage Profiler

The Stage Profiler decomposes each cold start into seven phases (see Fig. 4.2) and records a time stamp and duration per stage (23). Monitoring is intentionally simple and source specific, which means that the **first three stages are derived from the Kubernetes API** (pod conditions and container status timestamps), while **stages four through seven are recorded via OpenTelemetry spans** generated by the instrumented request path. This split ensures accurate ground truth for control-plane events and rich context for data-plane readiness.

Stage 1 — Scheduling (Kubernetes API)

Definition: Interval from pod creation to admission by the scheduler on a suitable target node (i.e., *PodScheduled=True*).

How monitored: Read `metadata.creationTimestamp` and the `PodScheduled=True` condition's `lastTransitionTime`. The difference results in a scheduling delay.

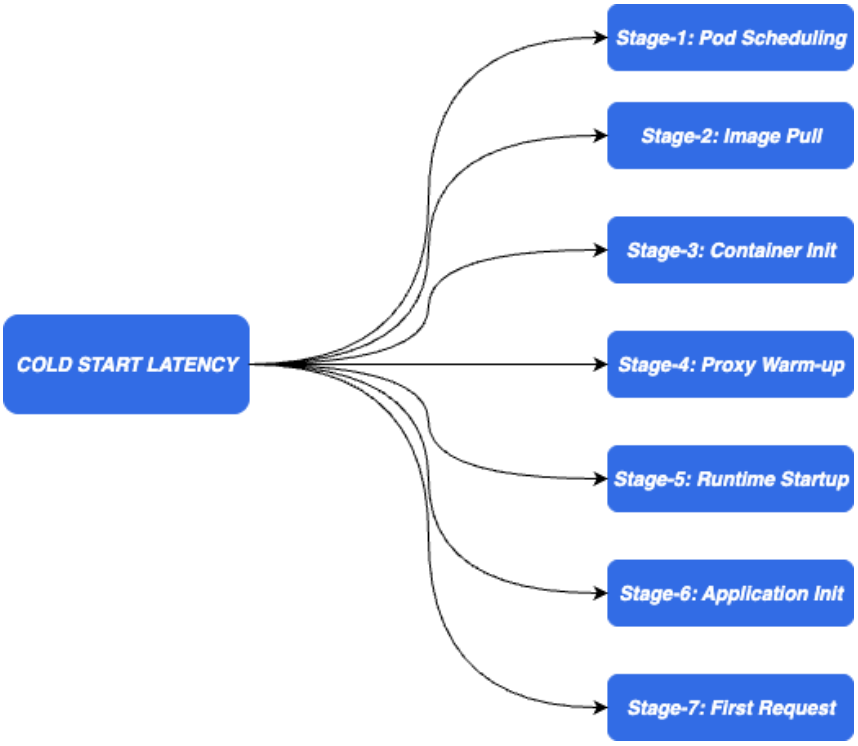

Figure 4.2: Classification of cold start latency into 7 Stages.

Stage 2 — Image Pull (Kubernetes API)

Definition: Interval during which container images are fetched and containers transition to *Running*.

How monitored: Use Pod events from the Kubernetes API. Look for event messages such as *"Pulling image ..."* and *"Successfully pulled image ..."*. The image pull interval is measured as the time between these two events, subtracting Stage 1 end time.

Stage 3 — Container Init (Kubernetes API)

Definition: Post–image-pull initialization until the pod becomes *Ready*/*ContainersReady*.

How monitored: Use the `Ready=True` (or `ContainersReady=True`) condition's `lastTransitionTime` minus the latest container `startedAt` from Stage 2.

Stage 4 — Proxy Warm-up (OpenTelemetry)

Definition: Time for the request path (ingress/activator/proxy) to become responsive, evidenced by the first successful response along the data plane.

How monitored: Active HTTP probes to the service endpoint were instrumented with an

OpenTelemetry span that began at the first probe attempt after pod readiness and ended at the first `200 OK`, by bracketing these two trace events, the proxy warm-up interval was obtained.

Stage 5 — Runtime Startup (OpenTelemetry)
Definition: Internal serving-runtime initialization (e.g., sidecar readiness) until the runtime can stably accept traffic.
How monitored: Readiness annotations emitted by the serving runtime (e.g., `queue-proxy`) were surfaced as trace events, an OpenTelemetry span covering the interval from the control-plane readiness milestone to the runtime's readiness event was recorded, thereby yielding the runtime-startup duration.

Stage 6 — Application Init (OpenTelemetry)
Definition: Application-level initialization (framework boot, caches, model load, etc.) between pod readiness and the first successful application response.
How monitored: The instant of pod readiness (from Stages 1–3) and the first `200 OK` observed in traces were linked via an OpenTelemetry span, by capturing this *Ready→200 OK* interval as a single traced segment, the application-initialization time was measured.

Stage 7 — First Request (OpenTelemetry)
Definition: Latency of the first measured request after warm-up, used as the canonical first-touch metric.
How monitored: Immediately after Stage 6, a single explicit request was generated and its end-to-end latency was recorded as an OpenTelemetry client span; this span constituted the first-request metric for cross-run comparison.

For every run, the profiler (i) emits per-stage timings for analysis and (ii) appends a JSON record with all stage timings and metadata to a log stream used by the optimizer.

4.1.3 Prompt Generation

The **Prompt Generation** module acts as the link between measurement and action. It converts diverse low-level data into a single, compact instruction that an LLM can process consistently. Without a structured prompt, optimization can become unreliable, difficult

to compare across runs and costly in terms of tokens (24). A schema-first template offers three benefits: (i) consistency in what information the model receives, (ii) safety through explicit guardrails, and (iii) cost control by limiting token use. This approach is similar to earlier LLM-based tuning systems that combine manuals, metrics and constraints before inference. (10) (25).

4.1.3.1 Inputs

The module takes in four inputs:

1. **Stage timings**: the seven per-stage durations and a minimal history (e.g., latest and previous runs), including a short bottleneck summary.

2. **Service YAML**: the current Knative `Service` manifest, reduced to actionable keys (e.g., `autoscaling.knative.dev/minScale`, `containerConcurrency`, CPU/memory requests/limits, selected probe parameters).

3. **Function/workload type**: a concise hint (e.g., *dependency-heavy*, *lightweight/CPU-light*, *DB-like/IO-bound*) that guides which levers to prioritize (image/initialization path, concurrency/target alignment, connection/IO posture).

4. **Node/cluster specs**: CPU cores, memory (GiB), ephemeral storage (GiB), mesh requirement flag, Knative version and an optional `max_replicas_hint`; used to ensure scheduling feasibility and capacity-aware tuning.

4.1.3.2 How the prompt is generated

All inputs are standardized and arranged into a fixed layout to reduce uncertainty and variation:

- **Objective**: Minimize cold-start latency by configuration only.

- **Hard Requirements**: (i)Config-only edits, (ii) respect node/cluster constraints, (iii) mesh-awareness.

- **Telemetry Block**: seven stage durations with units and a top-k bottleneck list (share of total).

- **Configuration Block**: current actionable keys and allowable ranges (explicit knobs the model may touch).

- **Constraints**: SLO/cost boundaries and scheduling feasibility hints.
- **Workload Type**: one of {dependency-heavy, lightweight, DB-like} to steer emphasis.
- **Output Contract**: "Return a minimal YAML patch (or compact full `Service`) as a single fenced code block; no prose."

This layout is model-independent, so the same prompt can be used to OpenAI's GPT, Claude's Sonnet or Google's Gemini through a plug-in adapter without changing content or order.

4.1.3.3 Why token budgeting

Long prompts increase both cost and latency, and they weaken the model's focus. Recent studies show that when context becomes too long, inference costs increase and accuracy can drop because attention weights are spread too thin across tokens (26). GPTuner showed that using compact prompts, which include only the most relevant metrics and configuration details, can achieve effective optimization at much lower token cost (10). Inspired by this principle, the module applies token budgeting to reduce unnecessary input size while preserving the essential context for decision making.

1. **Schema compaction**: The text is adjusted to follow a stable key order, and numeric values for stage times are rounded.
2. **Delta encoding**: The output expresses only the differences from the current manifest, with repeated or unchanged sections represented implicitly.
3. **Top-k focus**: The system highlights the largest contributing stages together with any stages above a fixed share threshold.
4. **Range summarization**: The text is simplified by expressing allowable values as intervals or enumerations instead of full prose.

These steps keep the prompt within the target budget while preserving the key context the LLM needs to create accurate patches.

4.1.3.4 Example output from the prompt generator

```
# Knative Config Optimization

Your job is to produce an UPDATED Knative Service YAML that improves cold-start latency
    ↪ by interpreting the provided stage timings, the current Service manifest and
    ↪ node/cluster information.

Use the provided inputs to reason about configuration changes.
Keep the output minimal and token-lean.

## Hard Requirements
- CONFIG-ONLY. Do not change app code, image contents or business logic.
- Stay within node budget. Use the provided node/cluster specs; avoid configs that cant
    ↪ schedule.
- Be mesh-aware. If a sidecar/mesh is not required, remove it; if required, keep it and
    ↪ tune around it.
- Minimal output: return only the YAML file.
- Prefer stability over overfitting. Make targeted adjustments.

## Inputs

### Node/cluster info (concise)
<node/cluster info>

### Function/workload type
- <workload-type>

### Stage timings (verbatim)
<stage-timings>

### Current Service YAML (verbatim)
<current-knative-yaml>


## Output contract
- Return only the YAML file (Knative Service manifest) in a single fenced code block.
```

4.1.3.5 Why this template matters.

A structured, workload-aware prompt turns telemetry and manifests into a short decision brief. It ensures safety by avoiding code changes, checks feasibility by fitting within node capacity and policy, and maintains parsimony by keeping the YAML minimal. The fixed schema also makes evaluations repeatable and comparable across workloads and LLM

backends, which is important for a plug-in optimizer. After it is created, the finalized prompt is forwarded to the Autonomous Optimizer.

4.1.4 Autonomous Optimizer

The **Autonomous Optimizer** is the decision-making core of the framework. It consumes the structured prompt from the Prompt Generation module and returns an updated Knative manifest *Service* designed to reduce cold-start latency using *configuration-only* changes. It consists of two parts. The first is a pluggable *LLM backend* that reasons over the prompt. The second is the resulting *manifest artifact* (YAML), which is applied to create a new Knative revision.

4.1.4.1 Scope of changes

All modifications are confined to the Knative configuration surface i.e., fields and annotations mentioned in the `Service` YAML. The optimizer does *not* alter the application code, the container images or the system architecture. This boundary focuses the approach on deployment-time levers that impact cold starts while preserving safety and reproducibility.

4.1.4.2 Plugin-based model interface

The LLM is treated as a replaceable module behind a common ‘prompt input, YAML out ‘ adapter. Different backends can be used interchangeably, for example *OpenAI ChatGPT* (27) , *Anthropic Claude (Sonnet)* (28), or *Google Gemini* (29) without changing surrounding components, allowing fair and repeatable comparisons between models. [1]

4.1.4.3 End-to-end flow

1. **Inference:** The optimizer submits the structured prompt to the selected LLM and receives *only* a YAML manifest as output (per the prompt contract).

2. **Apply:** The returned manifest is applied directly to Knative to produce a new *Revision*. The manifest and metadata are logged.

3. **Feedback & acceptance policy:** After a short evaluation window (e.g., a burn-in period or fixed request budget), the observability pipeline measures the cold-start

[1] In the prototype, a single API adapter was not implemented because of cost and the differences between provider APIs. The generator prompts were manually submitted to the chosen LLM, and the YAML patches returned were applied to create new Knative revisions.

metric L_{curr} for the new revision and compares it to the metric of the last accepted revision, L_{prev}. A tolerance τ defines the acceptance policy:

- **Accept (policy):** if $L_{\text{curr}} < L_{\text{prev}}$, mark the new revision as the *current best* and continue.
- **Exploration:** if $L_{\text{curr}} \geq L_{\text{prev}}$ but $L_{\text{curr}} \leq \tau$, keep the new revision in place for now. The framework may continue to explore from this state.
- **Revision (update):** if exploration shows poor results or the current latency exceeds the tolerance threshold, create a new revision with updated parameters and evaluate it.
- **Rollback:** if the revision performs worse than the tolerance, revert traffic to the last known good revision (the previous accepted one), restoring the stable configuration.

The exploration policy can be assigned with a configurable time interval (ranging from minutes to hours) that determines how long a revision is accepted before further exploration actions are taken. Similarly, the revision policy also includes a retry limit, which specifies the number of unsuccessful attempts allowed before triggering an automatic rollback.

This policy allows temporary exploration within the tolerance, while ensuring that poor revisions are either updated or rolled back to the best known configuration. Restricting edits to the Knative configuration layer keeps changes both safe and auditable, and the plugin design separates model choice from the system logic. The explicit policies for acceptance, exploration, revision, and rollback act as safeguards against regressions. Revisions that are only slightly worse are allowed to remain within the tolerance, but any candidate that performs worse than both the previous revision and the tolerance threshold is rolled back immediately.

5

Evaluation

This section evaluates the framework under continuous operation with changing conditions. The goal is to show how the framework holds cold start latency low and stable over time on real runs that include repeated scale from zero. The comparison point is a static baseline per workload that keeps one fixed configuration throughout the run on the same cluster and traffic pattern.

Three workload classes are used so that different cold start bottlenecks are covered. The classes are *lightweight*, *dependency heavy* and *network bound functions*. The cluster is a fixed Knative deployment on virtual machines provided by Continuum.

- **Lightweight functions** have small images and simple startup such as HTTP echo, JSON transformation, basic parameter checks and cache lookups.
- **Dependency heavy functions** have larger images and non trivial startup such as PDF processing, image conversion, ML model import or multiple library bootstraps. Latency is driven by image pulls and application initialization.
- **Network bound functions** call external data services such as SQL queries, key value store operations or authentication backed lookups. Latency is driven by connection setup, TLS handshakes, pool warmup and early I/O.

These three categories are not chosen arbitrarily but reflect bottlenecks commonly identified in prior serverless studies. Lightweight functions are the most frequent in production traces of cloud platforms (30). Dependency-heavy functions represent analytics and ML workloads, where cold starts are largely due to initialization costs (31). Network-bound functions capture services in which external I/O and TLS setup are the main limiting factors (32).

To measure these effects consistently across workloads, the framework employs an observability layer that combines the Kubernetes API and OpenTelemetry with Jaeger to record the seven stage timings for every cold start. The evaluation metric L is defined as the total cold start time from the scheduling to the first successful request, as introduced in Section 4.1.4.

All LLM backends are given the same prompt template and token budget to ensure fair comparison. To create genuine cold starts, scale-from-zero events are triggered at regular intervals during the hour. The figures show how each LLM backend performs within the framework and how the framework behaves under continuous operation with changing load.

5.1 Evaluation Setup

This section defines the environment and protocol used in all experiments. The setup consists of four parts:

5.1.1 Cluster VMs (Experiment Setup)

The experiments run on a Kubernetes/Knative cluster provisioned via Continuum across two virtual machines running Ubuntu 20.04, containerd 1.7 and Kubernetes v1.27 with Flannel CNI and Knative Serving v0.26. Observability is provided by the OpenTelemetry Collector and Jaeger (all-in-one), which record the seven-stage timings for each cold start. The cluster configuration is as follows:

Controller Node —

- CPU: 8 cores (capacity and allocatable)
- Memory: ∼31.36 GiB capacity, ∼31.25 GiB allocatable (overhead ∼0.1 GiB)
- Ephemeral Storage: ∼21.34 GiB capacity, ∼19.2 GiB allocatable (overhead ∼2 GiB)
- Role: Kubernetes control plane components (API server, scheduler, controllers, etcd).

Worker Node —

- CPU: 8 cores (capacity and allocatable)

- Memory: ∼31.36 GiB capacity, ∼31.25 GiB allocatable (overhead ∼0.1 GiB)
- Ephemeral Storage: ∼21.34 GiB capacity, ∼19.2 GiB allocatable
- Role: Executes Knative services and workloads (kubelet, kube-proxy, container runtime).

5.1.2 LLM backends

The optimizer uses three interchangeable LLM models through the same plugin interface, **ChatGPT-5**, **Claude Sonnet 4**, and **Gemini 2.5 Pro**. Each LLM receives the seven-stage timings from the profiler, the current Knative `Service` manifest, and the node/VM specifications (e.g., CPU, memory, and ephemeral storage). It then returns configuration-only updates to the manifest. This keeps the rest of the loop identical across models and cleanly isolates the effect of the LLM choice.

5.1.3 Procedure

For each workload and backend, a **60-minute** closed-loop run is conducted. A driver periodically and randomly triggers scale-from-zero to induce genuine cold starts throughout the window. After each accepted update, Knative creates a new revision and the profiler measures the resulting total cold start time L (from scheduling to the first successful request). All changes are limited to the `Service` manifest, while the application code and images remain the same. Baseline runs use a hand-tuned configuration under the same conditions for comparison.

5.2 Continuous optimization over time

The framework is evaluated under a continuous and dynamic workload. The objective is to assess performance as traffic and resource conditions change over time and to compare that behavior with a hand tuned baseline. The closed loop runs with three LLM backends, **ChatGPT 5**, **Claude Sonnet 4** and **Gemini 2.5 Pro**, using the same plug-in interface and the same inputs, namely the stage timings and the current `Service` manifest. Each model proposes configuration-only updates and accepted proposals create new Knative revisions that are measured again by the profiler. The baseline is tested under the same conditions.

This design supports two evaluations. First, whether the framework keeps cold start latency low and consistent as conditions change rather than only providing an initial drop. Second, whether the outcomes are consistent across models, meaning whether the choice of plug-in has a significant effect on stability over time. The figures report total cold start time over the run, with the baseline and each accepted revision clearly marked. Taken together, the results indicate how well the framework fits the use case and how it compares with the hand-tuned configuration in a dynamic environment.

5.2.1 Lightweight Functions

We evaluate lightweight functions under a continuous, dynamic workload where small configuration changes can noticeably shift cold-start behavior. The analysis begins with a hand-tuned baseline to see how it holds up as conditions change, and then contrasts those results with the LLM-driven loop. Figure 5.1 shows the baseline time series for a representative lightweight service. The run begins high at about **43 s**, then drops sharply before a small rebound to around **35 s**. Soon after, latency falls much further to roughly **15 s**, followed by a large spike, and then a long downward drift toward the best level seen in this trace at about **10 s**. From there the curve continues to move up and down rather than holding a steady band. This pattern shows that a careful baseline picked by an experienced engineer does not stay reliable once traffic and resources change over time. The configuration needs to adjust to current conditions (both the available resources and the shape of the workload) to keep cold-start latency low and stable. These observations motivate a continuous configuration-based optimization loop instead of a fixed one-time setup.

Figure 5.2 compares three LLM backends on a lightweight service under changing traffic. All three show a large initial drop in cold-start latency. **Claude Sonnet 4** achieves the strongest early effect, ahead of the others by a small margin that stays under five seconds. **ChatGPT-5** follows closely and **Gemini 2.5 Pro** is only slightly behind ChatGPT-5, often by about a second. The small gaps indicate that the framework makes effective use of the same context across models during the first phase.

As the loop continues, the framework keeps the cold start latency in a tighter and more stable range. Initially, **Gemini 2.5 Pro** drop the cold start latency from 48 **s** to 17.123 **s** (also observed at 17.174 **s**), a reduction of 30.877 **s** or $\approx 64\%$, **ChatGPT-5** drops from 48 **s** and falls to 16.980 **s**, a reduction of 31.020 **s** or $\approx 65\%$ and **Claude**

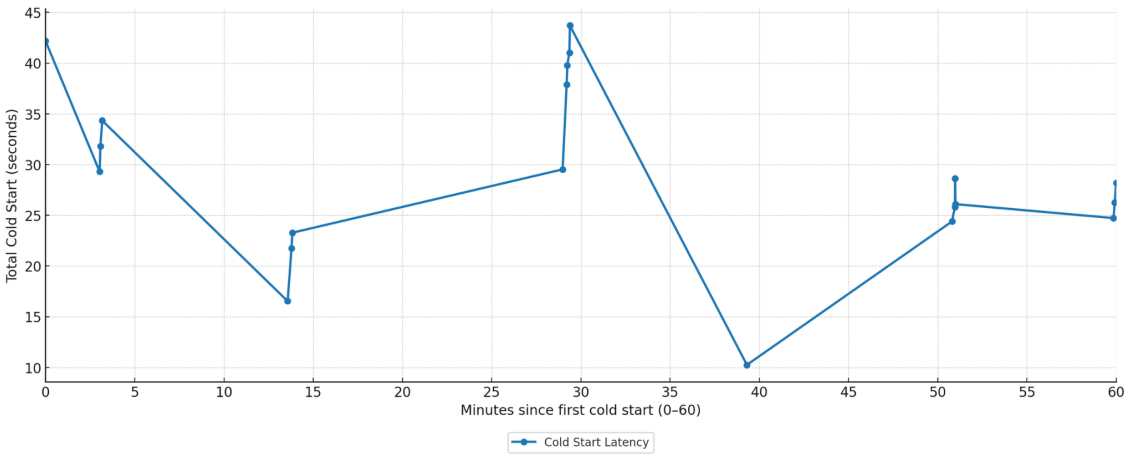

Figure 5.1: Baseline configuration on lightweight workload.

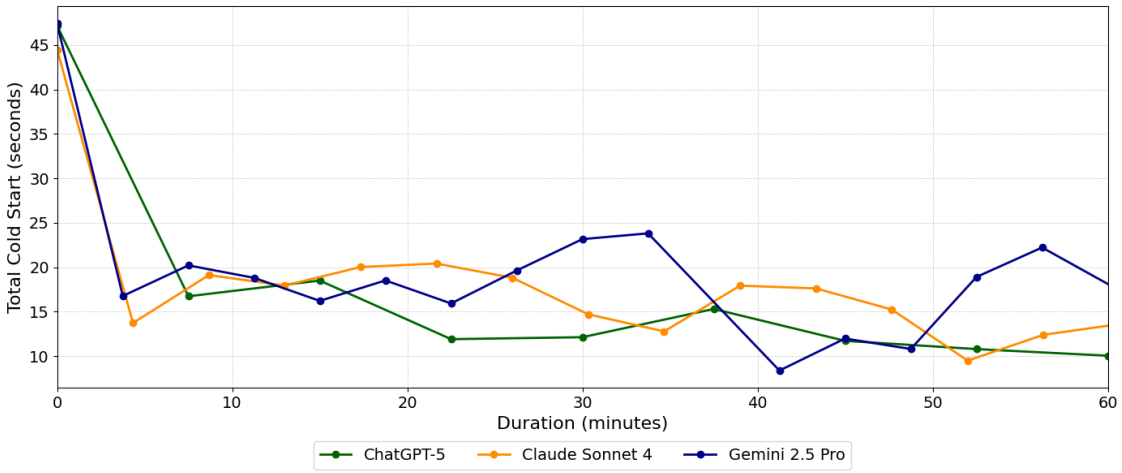

Figure 5.2: LLM-driven framework on lightweight workload.

Sonnet 4 drops from near ≈ 44 **s** and drops to 14.237 **s**, a reduction of ≈30 **s** or ≈ 68%. Over time ChatGPT-5 maintains the most consistent band and continues to edge down, with later gains below ≈ 20% and eventually in the low single digits ≈ 3–4%. Claude Sonnet 4 shows very strong points on a few occasions and briefly reaches lower levels than ChatGPT-5, yet these intervals are short and the overall gap between the two is small while ChatGPT-5 holds the low band for longer stretches. Gemini 2.5 Pro remains steady through roughly the first twenty minutes, then shows a rise that stays within tolerance, a subsequent revision produces a clear improvement and the curve settles again.

Across the lightweight workload the three backends perform closely and all outperform the baseline. Gemini meets tolerance goals but trails the other two slightly by the end. ChatGPT-5 finishes with the lowest and most stable after the loop stabilizes. Claude Sonnet$\tilde{4}$ delivers the best early reduction and remains stable, though not as low as ChatGPT-5 at the end. Overall, the framework adapts well in a dynamic setting and the choice of backend mainly changes how quickly and how far the pattern improves over time. The absolute gaps among the models are small—typically only a few seconds—and all runs stay within the acceptance tolerance for most of the window.

5.2.2 Dependency-heavy functions.

Figure 5.3 shows the baseline behavior for this workload under changing load. The trace starts around **50 s**, improves briefly, then rises and produces a large spike near **235 s**. Later it drops to the low **20 s** range, but does not stay there. It climbs again and moves between roughly **50** and **160 s** with repeated swings. These wide movements and long drifts indicate that a fixed, hand-tuned configuration cannot keep pace with changing conditions. In dependency-heavy services, image pulls and initialization cost shift with cache state, network conditions, and resource pressure, which makes the baseline unstable. Because these functions draw more CPU, memory, and network I/O during startup, a fixed hand-tuned plan is unlikely to remain efficient as the workload changes over time.

Compared to the baseline, Figure 5.6 shows that all three models quickly compress cold starts into a much lower band and then handle disturbances with small and bounded swings. **Gemini 2.5 Pro** drops from 50.000 **s** to 8.000 **s**, a reduction of 42.000 **s** or $\approx 84\%$. It experiences a mid-run rise between seven and thirty minutes, but the increase remains within the acceptance tolerance and is corrected without rollback. After this period Gemini stabilizes again and remains consistently low until the end of the experiment.

ChatGPT-5 shows the smallest initial reduction, moving from 40.000 **s** to 21.000 **s**, a decrease of 19.000 **s** or $\approx 48\%$. However, over time its performance improves further and it maintains the lowest and most stable latency across the continuous workload scenario, converging to a narrow band well below baseline.

Claude Sonnet 4 reduces from 43.500 **s** to 29.863 **s**, a reduction of 13.637 **s** or $\approx 31\%$. Although it later drops further, Claude exhibits a mid-run spike that briefly

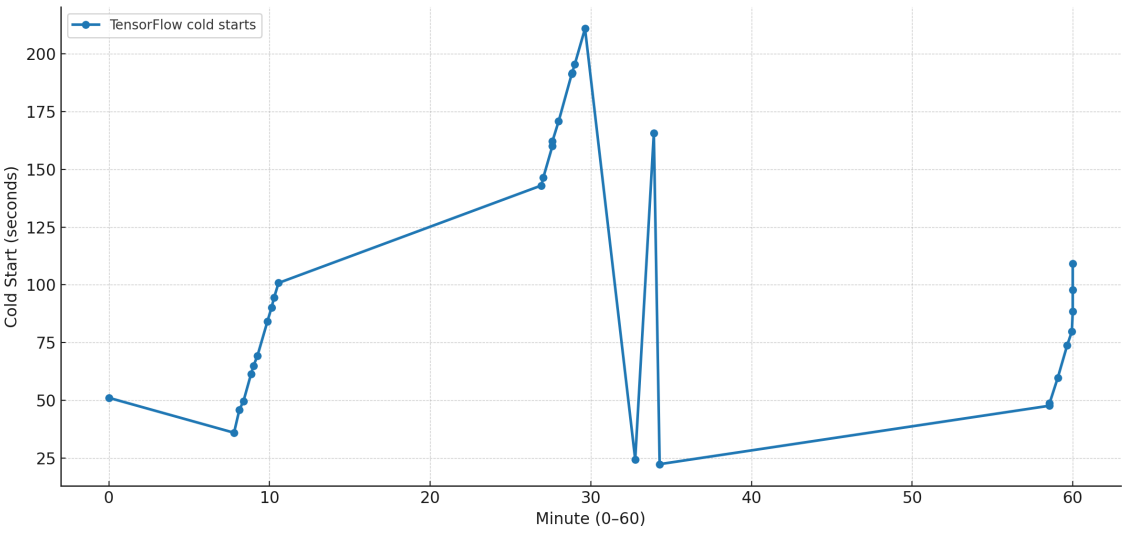

Figure 5.3: Baseline on dependency-heavy workload.

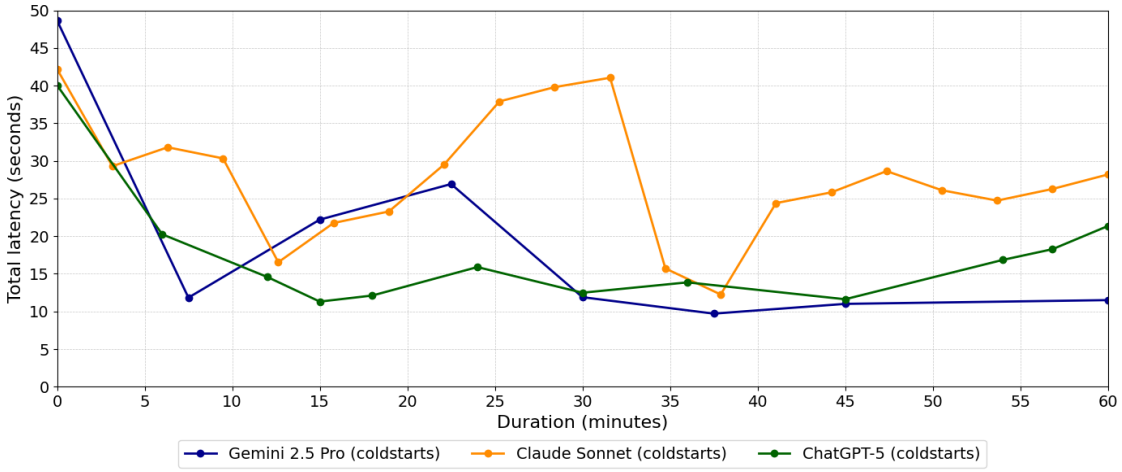

Figure 5.4: LLM-driven framework on dependency-heavy workload.

pushes latency back near the initial cold-start level. After a corrective revision is applied, it returns to the lower range and holds near 30 **s**, but its trajectory remains less predictable compared to Gemini and ChatGPT.

All three models make the dependency-heavy service far more consistent than the static baseline. ChatGPT-5 finishes with the lowest and most stable across the window. Gemini delivers the largest initial drop and, after tolerating a mid-run rise, maintains a competitive steady state. Claude shows the smallest initial reduction

and its mid-run instability makes it less reliable, even though it stabilizes later within tolerance. In steady state, the gap between the three is reduced to a few seconds, which suggests that the closed-loop policy accounts for much of the sustained gain regardless of the specific model. Taken together, the traces show a common pattern: rapid early drop, occasional mid-run correction, and settling down to a narrow band well below baseline, indicating that the feedback loop is the primary driver of sustained cold-start reduction.

5.2.3 Network Bound Function

Figure 5.5 shows the trace begins around 47 s, dips once to the mid 20 s around 16 to 17 minutes, and then trends upward for the remainder of the hour, finishing close to 90 s. This early dip followed by a long rise is typical for DB like services because cache warmups fade, connection pools churn, and disk and network contention accumulate. The result is a steadily worsening cold start that a fixed manually optimized manifest cannot counteract, which shows that static tuning is brittle under changing load.

Compared to the baseline, Figure 5.6 shows that all three models quickly compress cold starts into a much lower band and then handle disturbances with small and bounded swings. **Gemini 2.5 Pro** drops from 50.123 **s** to 14.320 **s**, a reduction of 35.803 **s** or $\approx 71\%$. **Claude Sonnet 4** moves from 54.350 **s** to 14.6845 **s**, a reduction of 39.6655 **s** or $\approx 73\%$, and remains within 0.3645 **s** of the Gemini level. **ChatGPT 5** reduces from 50.000 **s** to 30.549 **s**, a reduction of 19.451 **s** or $\approx 39\%$, and then maintains the most consistent band across the hour, holding near ≈ 30 **s** and occasionally reaching as low as ≈ 10 **s**. Gemini shows a brief bump between fifteen and seventeen minutes and around twenty minutes a configuration revision is attempted because the current latency exceeds the tolerance. The change does not help and the rollback policy restores the previous setting. After the rollback Gemini settles again and stays low and stable through the end. Claude follows a similar pattern with one late spike near fifty three to fifty four minutes that briefly approaches ≈ 100 **s**. After a revision it returns to the low range and holds near ≈ 30 **s**.

When comparing the three backends across workloads, each shows its own strength. In the lightweight case, Claude Sonnet 4 gave the sharpest early drop, but ChatGPT-5 held the lowest and most stable levels over time. For dependency-heavy functions, Gemini 2.5 Pro produced the largest initial reduction and recovered quickly from mid-run rises, while ChatGPT-5 again ended with the most stable long-term trend. In

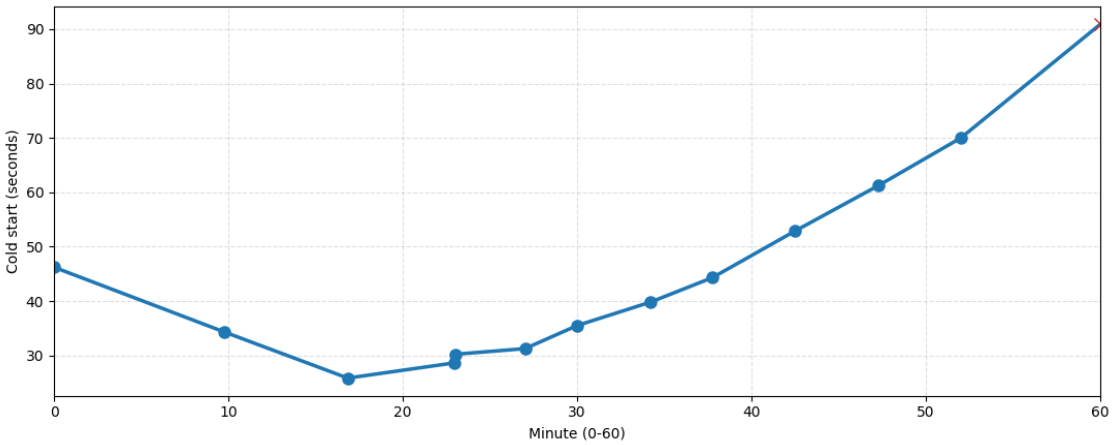

Figure 5.5: Baseline Configuration on network bound workload.

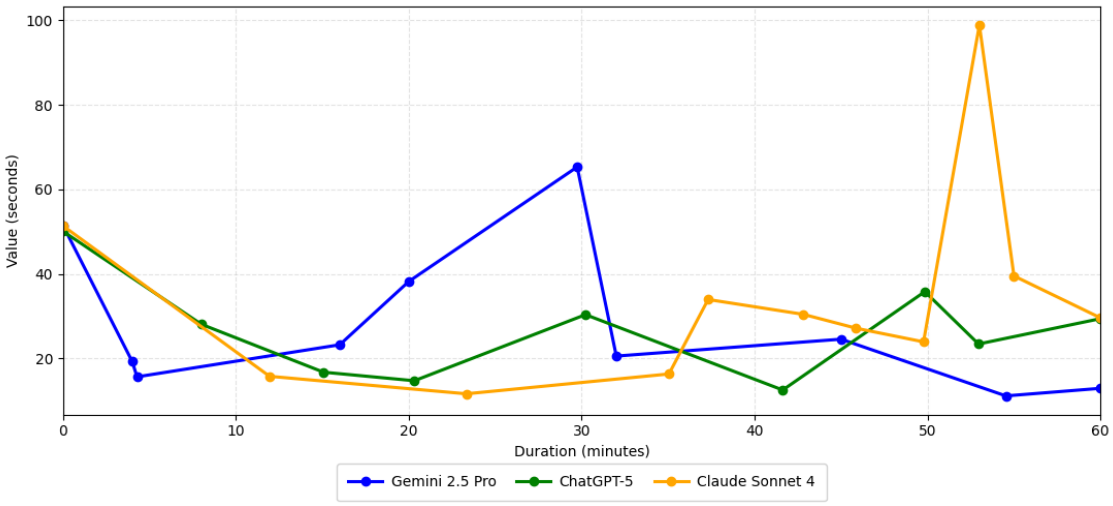

Figure 5.6: LLM-driven framework on network bound workload.

the network-bound workload, Claude and Gemini both pushed latency down into the mid-teens and stayed close together, but ChatGPT-5 maintained the most consistent range once the run settled. Taken together, ChatGPT-5 stands out for stability, Claude Sonnet 4 for its strong early reductions, and Gemini 2.5 Pro for its aggressive initial drops with solid recovery.

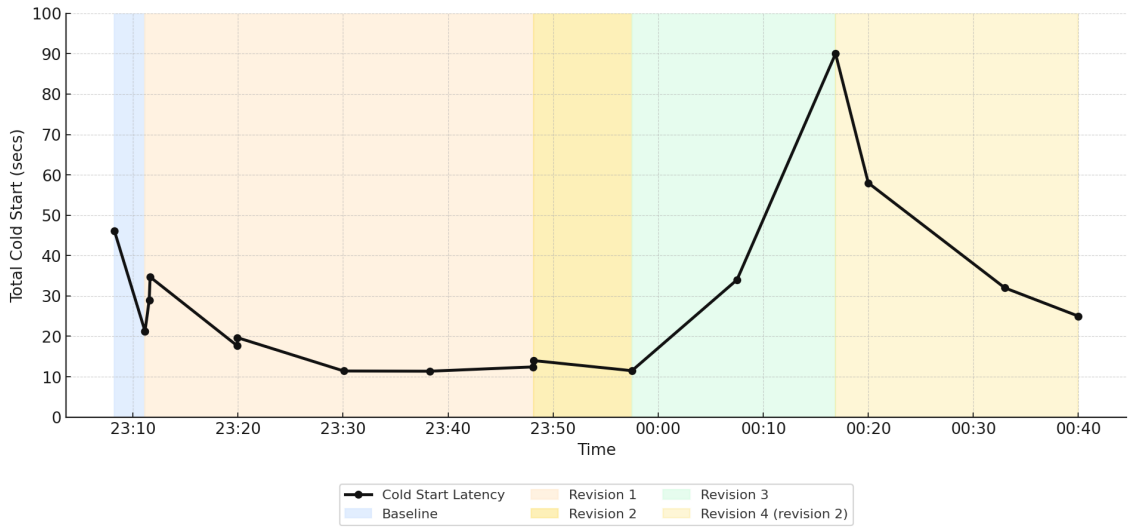

Figure 5.7: Rollback policy on dependency-heavy workload

5.2.4 Evaluation of the feedback loop and rollback policy

A separate experiment is included to show, in a best-case run, how the feedback loop and the rollback policy behave (Figure 5.7). The run uses the *dependency-heavy* workload for one hour under the same settings as the continuous experiments, with repeated scale-from-zero so that genuine cold starts occur. The LLM backend is **ChatGPT-5**. The observability pipeline combines the Kubernetes API and Open-Telemetry with Jaeger and records the seven stage timings. To make revision events visible within a single run, the exploration timer for proposing a new configuration is set to $\approx 10\,\text{min}$.

The first accepted change yields an improvement of $\approx 53.5\%$, and the series remains steady afterward with only a small bump. After a period of stable operation, the exploration timer fires and a forced attempt is made to test whether a higher level can be reached, creating *revision three*. The candidate underperforms and cold-start latency rises sharply to $\approx 90\,\text{s}$, so the rollback policy engages and *revision four* restores the last known good configuration from *revision two*. Once restored, the latency returns to the earlier low range and stays there for the rest of the run.

This example shows why controlled exploration together with automatic rollback is necessary under changing conditions. The framework can probe for additional gains on a fixed interval, detect regressions quickly with guard metrics and tolerance

checks, and limit impact by returning to the prior stable configuration. In practice, this keeps the service close to the low range achieved after the first improvement while still allowing safe attempts to discover a better configuration for the current window.

6

Future Direction

The current framework reduces cold start time and keeps a stable cold start latency under changing load. It operates as a simple closed loop that proposes configuration changes, monitors guard metrics and rolls back when needed. The loop generalizes across workload types and remains well below the static baseline for most of the hour long runs. The LLM has black box characteristics and its internal reasoning is opaque. Structured prompting, an explicit output requirement and schema-guarded fields establish a clear boundary around the model, ensuring proposals remain within policy and produce the configuration patches the framework requires. Tolerance checks and a revision log provide practical control over volatility. These results show that the framework delivers meaningful gains without any changes to application code or images, while there is a clear room to improve reliability and efficiency. The following outline future work that can strengthen and extend the framework.

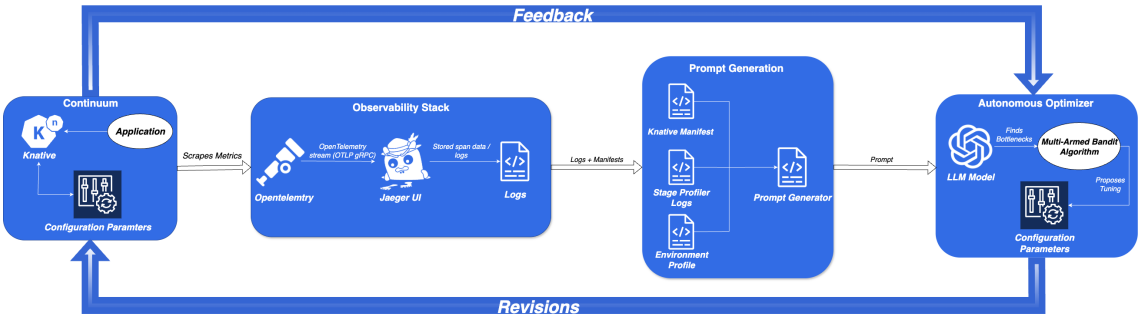

Figure 6.1: Framework architecture with a Multi-Armed Bandit Algorithm.

6.1 Machine learning model for optimization

As future work, the optimizer can be replaced with a trained model. The closed loop and guard metrics stay the same, as shown in Figure 6.1. Only the optimizer changes. This option fits cases that need offline execution, stronger security, and more predictable behavior learned from past runs. The model is trained and checked offline, reports confidence scores, and can skip a change when risk is high. This makes decisions easier to review and control.

To show this idea, a *multi-armed bandit (MAB)* is used instead of the LLM (33). The bandit is lightweight, learns from feedback with simple rules, and balances trying new options with using good ones. Each arm is a configuration option for a given scenario. The framework uses a two-level bandit: **Level 1** picks the workload type for the next step, **Level 2** picks the stage with the biggest bottleneck in that workload and then chooses an arm from the allowed patch options. The selected change is applied as a new Knative `Service` revision. The framework then computes a reward and updates the arm values, so later runs are matched with better configurations based on what was learned.

This MAB uses Thompson Sampling. In a continuous setting, frequent revisions and too little data per revision sometimes led to uneven behavior because the bandit did not have enough evidence. Even so, after several runs the bandit started to settle, and the results below show improvements over the baseline within the same architecture (Figure **??**).

In dependency-heavy workloads, the multi-armed bandit improves total cold start by about 50%. It starts weaker in the first three stages and then performs well in the later stages as it settles on stronger settings. It needs several trials to discover a

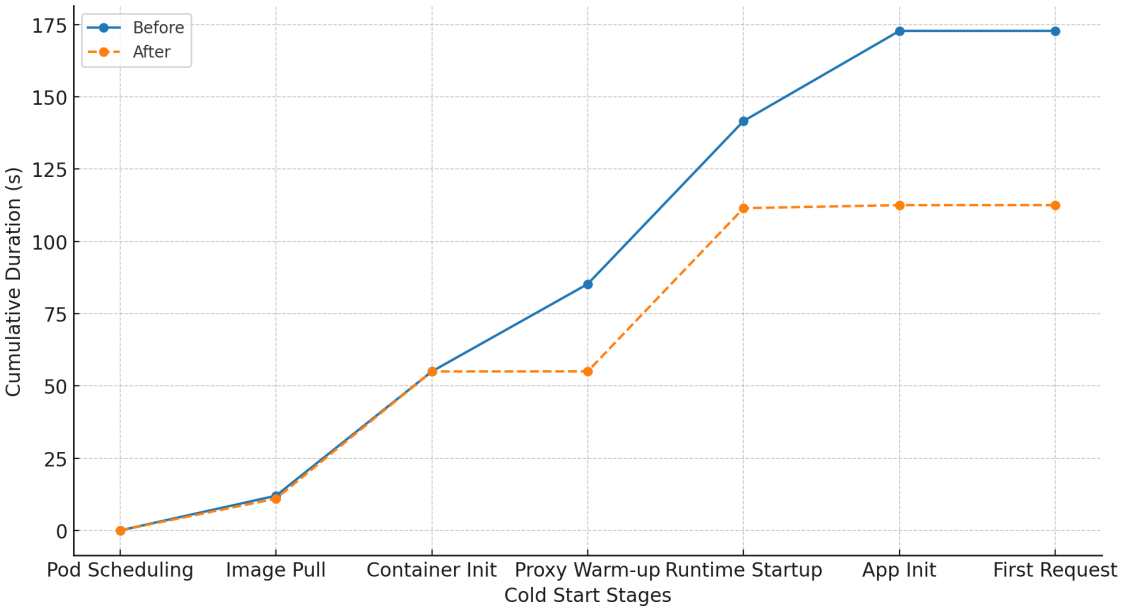

Figure 6.2: MAB-based tuning on lightweight workload (cold-start stages)

good configuration, which is expected for a simple explorer and this is the workload where the machine learning optimizer works best, see Figure 6.3. In the lightweight functions, the first three stages behave similarly to the baseline. After that it shows clear gains over the baseline across stages, but the gains are smaller when compared to the dependency-heavy case. This pattern indicates that the model is effective while still leaving room for improvement, Figure 6.2. In the network-bound functions, the overall gain is about 10% with a best case near 15%, Figure 6.4. The improvement is smaller than in the other workloads. Stage 1 matches the baseline, Stage 2 is better and Stage 3 trails the baseline. Later stages stay below the baseline but with smaller margins. These outcomes show that the bandit is a workable foundation and they suggest strong potential for a machine learning optimizer that learns offline with richer features and they also show why a learned model can be preferred over an LLM which remains a black box in this setting.

Overall, these findings highlight that the multi-armed bandit, while simple, plays an important role in establishing a solid foundation for the framework. It demonstrates that even lightweight online exploration can uncover meaningful gains and it motivates the shift towards more advanced machine learning–based optimizers that can

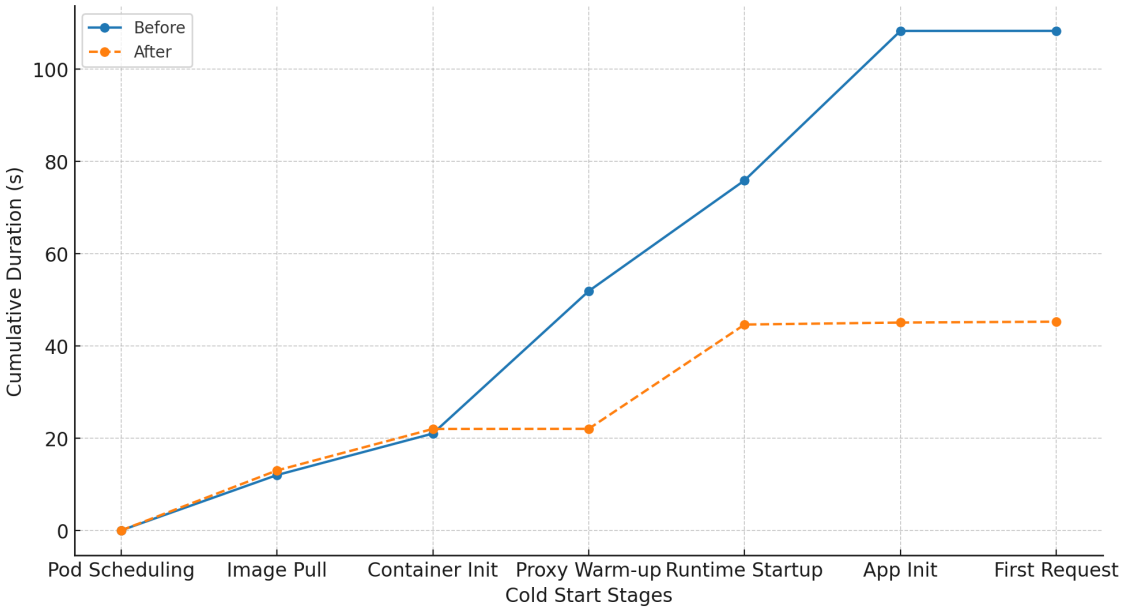

Figure 6.3: MAB-based tuning on dependency-heavy workload (cold-start stages)

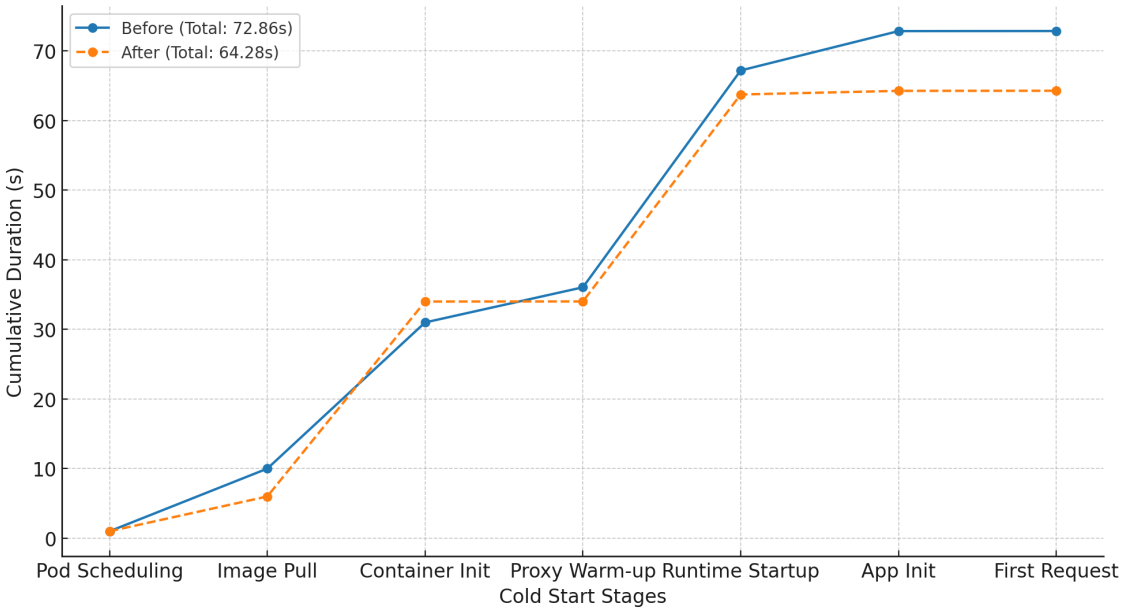

Figure 6.4: MAB-based tuning on network bound workload (cold-start stages)

leverage richer state and historical data. This progression shows why bandit-style exploration provides the groundwork for a scalable and future-proof optimization strategy.

6.2 Beyond Knative configuration

The focus at present is on Knative service fields, but the same approach could easily be extended beyond Knative to include platform-specific and infrastructure-level configurations. In practice, this means that optimization is not limited to annotations within the Knative `Service` manifest, but could also include settings exposed by cloud providers such as AWS, Azure or Google Cloud. Modifying parameters at these levels—such as autoscaling policies, runtime configurations, registry and storage behavior, networking or DNS resolution and cache or warm pool management—can influence key cold start stages including scheduling, image pull, container initialization and the first request. In addition, infrastructure-oriented changes like adjusting VM-level specifications (e.g., CPU, memory or topology) can shift the readiness baseline further, reducing total cold start time. Several of the heavier stages are in fact more constrained by platform or infrastructure limits than by Knative-level service parameters, which makes expanding the scope both logical and impactful. Since the LLM remains a black box, the framework gains more scope to focus on exposing clearer and more auditable levers at these additional layers, enabling stronger optimization not only at the Knative level but also across the broader platform and infrastructure stack.

6.3 Better rollback policy

In future work, the rollback mechanism could be extended with a more structured delivery strategy such as A–B or blue–green deployments (34)(35). In the current system, stage timings are monitored after each new configuration and a rollback is triggered if performance becomes worse. Although this ensures safety, adopting rollout styles such as A-B or blue-green would make the behavior more predictable and easier to manage in production (36). These methods would allow candidate revisions to be tested on a small slice of traffic or in parallel with the current version before shifting all traffic. Shadow traffic could also be explored as a way to evaluate

new revisions without exposing users to potential regressions (37). Moving in this direction would make rollback safer, reduce disruptions and increase the reliability of continuous optimization.

6.4 Energy aware cold start tuning

The current framework optimizes for latency and does not make energy or cost a direct goal. A small extension can add simple measurements so that each change is judged by both total cold-start time and resource use (38). Examples include CPU time spent per cold start, average idle memory kept and an optional carbon-intensity label for the time window. Such measurements are already being exposed by cloud providers, for example through the Google Cloud Carbon Footprint tool (39). With these measurements in place, the acceptance rule can prefer changes that reduce latency without increasing these measurements or keep latency within tolerance while lowering them. The prompt bundle can also pass a small budget hint so the optimizer suggests changes that save work during quiet periods. The expected impact is less idle waste, clearer reporting of cost and energy next to latency and a stable low-latency band achieved with fewer resources. This is particularly useful in production environments where cost or a fixed budget matters, because it makes energy and spend part of the decision rather than an afterthought.

7

Conclusion

Cold-start latency continues to be one of the most long-standing problems in serverless computing, limiting the use of these platforms for latency-sensitive applications. This thesis presented a practical framework that reduces cold-start delay in Knative by combining stage-aware profiling, structured prompt generation and large language model (LLM) driven configuration tuning. Static approaches often fail when workloads change, but this framework adapts continuously and keeps latency low without requiring changes to the code or images.

The contributions of this work lie in showing that configuration-only optimization is sufficient when paired with fine-grained observability and safe automation. By decomposing cold starts into seven measurable stages, the framework pinpoints the dominant bottlenecks and translates them into structured prompts for LLM backends. The optimizer proposes targeted YAML patches, applies them as new revisions, and evaluates their effect in real time. A rollback policy ensures that regressions are quickly corrected, so performance does not suffer for long.

The evaluation across three workload classes (lightweight, dependency heavy and network bound) demonstrated consistent improvements compared to static baselines. In measurable terms, the framework achieved cold-start reductions of up to **65–84%**, reducing delays from several tens of seconds to just over ten seconds. Just as importantly, it maintained stable latency bands over time rather than fluctuating as baselines did. The choice of LLM backend influenced the pattern of improvement, Claude Sonnet4 delivered the fastest early drops, ChatGPT-5 maintained the most stable trajectory, and Gemini2.5 Pro provided competitive results after mid-run cor-

rections, with overall differences being small in margin. Despite these differences, all three remained within acceptance tolerances and outperformed static configurations.

These findings can be summarized as follows:

- Stage-level profiling exposes bottlenecks clearly and enables targeted improvements.
- Structured, schema-guarded prompts make LLM optimization safe and repeatable.
- Configuration-only changes are powerful enough to yield large reductions without touching the application code or images.
- The feedback loop with rollback policy is the critical driver of sustained improvements, ensuring resistance to regressions.
- The differences between LLM backends are small, and the framework works well across models and workloads.

Together, these results show that black-box models like LLMs can be made reliable and effective when constrained within a well-defined optimization loop. The framework consistently adapts to changing runtime conditions and delivers performance improvements that static tuning cannot sustain.

Looking forward, this thesis also identified several directions for this solution. Replacing the LLM with a learned optimizer such as a multi-armed bandit or an offline-trained model could improve predictability, reduce costs and enhance auditability. Broadening the scope of the optimization beyond Knative manifests to include platform or infrastructure-level configurations would address additional stages. More improved rollout strategies, including A/B or blue-green deployments, could make revisions safer in production. Finally, integrating energy and cost awareness into the framework would align the framework with the sustainability and budget concerns that matter in real deployments.

In conclusion, this work demonstrates that a combination of observability, structured reasoning and guarded automation provides a practical path to mitigating cold-start latency in serverless computing. The framework not only significantly reduces latency, but also gracefully adapts to workload variability, offering both immediate impact and a foundation for future adaptive optimization systems in cloud native environments.

References

[1] Amazon Web Services. **Announcing AWS Lambda**. `https://aws.amazon.com/blogs/aws/introducing-aws-lambda/`, 2014. 1

[2] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Samuel Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Philippe Suter, and Olivier Tardieu. **Serverless Computing: Current Trends and Open Problems**. In *Research Advances in Cloud Computing*, pages 1–20. Springer, 2017. 1

[3] Eric Jonas, Jonah Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Kartik Khandelwal, Qifan Pu, Sameer Shankar, Shivaram Venkataraman, Benjamin Recht, and Ion Stoica. **Cloud Programming Simplified: A Berkeley View on Serverless Computing**. *arXiv preprint arXiv:1902.03383*, 2019. 1

[4] Muhammed Golec, Eyuphan Yilmaz, and Ibrahim Ali Dogru. **Cold Start Latency in Serverless Computing: A Systematic Review, Taxonomy, and Future Directions**. *arXiv preprint arXiv:2310.08437*, 2023. 1, 3, 8

[5] Mohammad Ebrahimi et al. **A Review on Mitigation of Cold Start Latency in Serverless Platforms**. *Journal of Cloud Computing*, 2024. 3, 8

[6] Wenguang Lin and Alex Glikson. **Mitigating Cold Starts in Serverless Platforms: A Pool-Based Approach**. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, pages 1–13, 2019. 3

[7] Dmitrii Ustiugov et al. **REAP: Reducing Cold Start Latency of Serverless Applications via Snapshotting and Prefetching**. In *Proceed-*

ings of the ACM European Conference on Computer Systems (EuroSys), pages 538–554, 2021. 3, 9

[8] Amazon Web Services. **AWS Lambda Cold Start and Warmup Strategies**, 2022. `https://aws.amazon.com/lambda/`. 3

[9] Thomas Rausch et al. **Pagurus: Dependency-Aware Container Reuse for Serverless Computing**. In *Proceedings of the 16th European Conference on Computer Systems (EuroSys)*, pages 29–44, 2021. 3

[10] Yong Zhang et al. **GPTuner: LLM-Guided Automatic Database Tuning**. In *Proceedings of the ACM SIGMOD Conference*. ACM, 2024. 4, 9, 15, 16

[11] Wei Jiang et al. **λ-Tune: LLM-Driven Database Configuration Generation for Serverless Workloads**. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. VLDB Endowment, 2023. 4, 9

[12] Rui Zhou et al. **SlsDetector: Using LLMs to Detect and Explain Misconfigurations in Serverless Systems**. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*. ACM, 2023. 4, 9

[13] Chen Wang et al. **LLMTune: Large Language Models for Automated Knob Tuning in Databases**. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 2024. 4

[14] Dana Van Aken, Andrew Pavlo, et al. **OtterTune: Automatic Database Management System Tuning Using Machine Learning**. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 631–645. ACM, 2017. 4

[15] Valentin Stoilov, Philipp Wiesner, and Jörg Domaschka. **Continuum: Towards a Benchmarking Framework for Cloud-Native Applications on Virtual Machines**. In *Proceedings of the 23rd International Middleware Conference (Middleware '22)*, pages 379–390. ACM, 2022. 6, 10

[16] Zhengchun Luo, Zhihong Li, and Calton Pu. **Characterizing Serverless Platforms: A Large-Scale Empirical Study of Knative**. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE '21)*, pages 101–112. ACM, 2021. 8

[17] Zhengchun Xu, Zhihong Li, and Calton Pu. **Characterizing Serverless Platforms: A Large-Scale Empirical Study of Knative**. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 101–112, 2021. 8

[18] Paolo Arcaini, Elvinia Riccobene, and Patrizia Scandurra. **Modeling and Analyzing MAPE-K Feedback Loops for Self-Adaptation**. In *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 13–23, 2015. 10

[19] Cloud Native Computing Foundation. **OpenTelemetry: Observability Framework for Cloud-Native Software**. `https://opentelemetry.io/`, 2021. Accessed: 2025-08-18. 10, 12

[20] Cloud Native Computing Foundation. **Jaeger: Open Source, End-to-End Distributed Tracing**. `https://www.jaegertracing.io/`, 2017. Accessed: 2025-08-18. 10, 12

[21] Matthijs Jansen, Linus Wagner, Animesh Trivedi, and Alexandru Iosup. **Continuum: Automate Infrastructure Deployment and Benchmarking in the Compute Continuum**. In *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering (ICPE '23 Companion)*, pages 181–188. ACM, 2023. 11

[22] The Kubernetes Authors. **Resource metrics pipeline**, August 2024. Last updated August 31, 2024. 12

[23] r0han0908. **stage_profiler: lightweight instrumentation library for serverless functions**. `https://github.com/r0han0908/stage_profiler`, 2025. Commit as of July 13, 2025. 12

[24] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. **Chain-of-Thought Prompting Elicits Reasoning in Large Language Models**. In *Proceedings of the 36th International Conference on Neural Information Processing Systems (NeurIPS '22)*, 2022. 15

[25] Takeshi Kojima, Shixiang Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. **Large Language Models are Zero-Shot Reasoners**.

In *Proceedings of the 36th International Conference on Neural Information Processing Systems (NeurIPS '22)*, 2022. 15

[26] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Percy Liang, and Christopher D. Manning. **Lost in the Middle: How Language Models Use Long Contexts**. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (ACL '23)*, 2023. 16

[27] OpenAI. **ChatGPT**, 2025. Accessed: Aug 23, 2025. 18

[28] Anthropic. **Claude Sonnet 4**, 2025. Accessed: Aug 23, 2025. 18

[29] Google DeepMind. **Gemini 2.5 Pro**, 2025. Accessed: Aug 23, 2025. 18

[30] Mohammad Shahrad, Jonathan Balkind, Xu Weng, Prateeksha Gill, Tim Hadjilambrou, Henry Lim, Runjie Tang, Rodrigo Fonseca, and Margaret Martonosi. **Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider**. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*, pages 205–218. USENIX Association, 2020. 20

[31] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. **From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers**. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC '19)*. USENIX Association, July 2019. Available as open-access PDF. 20

[32] Simon Eismann, Joel Scheuner, Erwin van Eyk, Manuel Schwinger, Johannes Grohmann, Nikolas Herbst, and Samuel Kounev. **A Review of Serverless Computing Performance Characteristics**. *ACM Computing Surveys*, **53**(3):1–39, 2021. 20

[33] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. **Finite-time Analysis of the Multiarmed Bandit Problem**. *Machine Learning*, **47**(2-3):235–256, 2002. 32

[34] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.* Addison-Wesley, 2010. 35

[35] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. *Site Reliability Engineering: How Google Runs Production Systems.* O'Reilly Media, 2016. 35

[36] The Kubernetes Authors. **Strategies for Managing Rollouts**, 2025. Accessed: Aug 23, 2025. 35

[37] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. **Large-Scale Cluster Management at Google with Borg**. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys '15)*, pages 1–17. ACM, 2015. 36

[38] Simon Eismann, Pooyan Jamshidi, Johannes Grohmann, Nikolas Herbst, and Samuel Kounev. **Sustainable Serverless: Cloud Sustainability from a Serverless Perspective**. In *Proceedings of the 22nd International Middleware Conference (Middleware '21)*. ACM, 2021. 36

[39] Google Cloud. **Carbon Footprint**, 2022. Accessed: Aug 23, 2025. 36