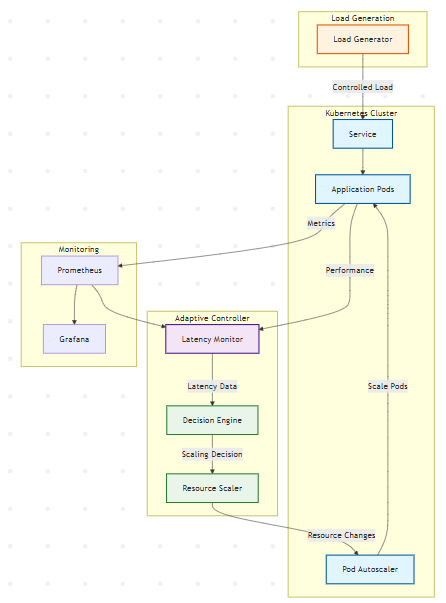
# Initial Architecture



### High-Level Architecture Overview

1. Initial Deployment -> 2. Stability Phase -> 3. Stress Phase -> 4. Adaptive Vertical Scaling -> 5. Monitoring Loop

### Stability Phase – Base Workload Discovery

To automatically discover the most stable request rate (R₀) that our application can handle without overloading the system and while maintaining a low, steady latency. The latency and CPU usage observed at this point form our performance baseline (L₀).

#### Why This Matters:

* Applying too low a load wastes resources (underutilization).
* Applying too high a load causes latency spikes or failures.
* This phase helps find the "sweet spot" for the application in its current resource envelope (CPU/memory requests).

#### Algorithm: Auto-Selecting the Stable Load

1. **Start with a Controlled Initial Load**

* Initialize request rate at 1 request per second (RPS).
* Ensure that this is well below what the service can handle.
* Allow the system to stabilize for ~30 seconds to 1 minute.
* Measure and log:
  + p95\_latency(t)
  + CPU\_Usage(t)
  + Memory\_Usage(t)

1. **Ramp-Up Loop (Iterative Load Increase)**

* Loop Logic(Pseudo Code)

R = 1 # start RPS

ΔR = 1 # increment in each step

interval = 30 seconds # time to observe metrics per step

while True:

apply\_load(R)

wait(interval)

L = get\_p95\_latency()

cpu = get\_cpu\_usage\_percent()

# Check for breaking conditions

if (L - previous\_L) / ΔR > LATENCY\_SENSITIVITY\_THRESHOLD:

break

if cpu > MAX\_CPU\_THRESHOLD:

break

previous\_L = L

R += ΔR

* The division:
  + (L - previous\_L) / ΔR represents the rate of change of latency with respect to request rate — essentially, the "latency gradient".
  + It tells us, "How much does the latency increase per each additional request per second (RPS)?"
  + Why we use it:
    - To Detect Performance Saturation: As we increase RPS, the service initially handles it well (small latency changes). But when we approach resource limits (CPU/memory/thread pool), even a small increase in RPS causes a big jump in latency. This formula helps detect that inflection point.
    - Gradient Helps Compare Across Services: By calculating the rate of change rather than the absolute change, we make the condition:
      * Scalable across slow and fast services
      * Independent of absolute latency numbers
      * Easier to compare
    - It's Like a Derivative (Δy/Δx): It’s conceptually the first derivative of latency w.r.t. request rate: How fast is latency growing when load increases? If this slope (gradient) becomes too steep, we stop ramping.
  + Why Not Just Use Latency Alone: Because raw latency can fluctuate, and a high latency at one point might not be a trend. This gradient checks whether latency is becoming sensitive to load, not just high.
  + LATENCY\_SENSITIVITY\_THRESHOLD
    - We can define the LATENCY\_SENSITIVITY\_THRESHOLD using our initial latency (L’) by expressing it as a percentage of acceptable latency increase per additional RPS. This method is adaptive and scales with the responsiveness of our service.
    - LATENCY\_SENSITIVITY\_THRESHOLD = α × L’, Where: L’ = Initial latency (e.g., p95 latency under stable initial load), α = Sensitivity factor (usually between 0.03 and 0.10)

| Service Type | Suggested α | Meaning |
| --- | --- | --- |
| Real-time Systems | 0.03 (3%) | Sensitive to latency increases |
| Web APIs / SaaS | 0.05 (5%) | Moderately sensitive |
| Async/Background APIs | 0.07–0.10 | More tolerant |

* Why This Works:
  + It adapts to both fast and slow services.
  + Makes our ramp-up logic generalizable to different microservices.
  + Prevents arbitrary hardcoded thresholds that may not fit all services.
* How to calculate initial latency fro the threshold definition:
  + To find the base latency (L’) initially, you need to follow a controlled, data-driven process right after deployment and before any optimization or scaling begins.
  + How to Find Initial Latency (L’)
    - Start the Service with Minimal Load: Deploy the application with default CPU/memory (e.g., 250m CPU, 256Mi RAM). Set the initial request rate to something low, like 1 RPS.
    - Warm Up the Service: Let the service run for ~2–5 minutes(This allows JIT warmup (for JVM-based apps), caching, internal thread pool startup, etc.)
    - Start Monitoring p95 Latency: Using Prometheus(histogram\_quantile(0.95,rate(http\_request\_duration\_seconds\_bucket[1m])))
    - Ensure Latency is Stable: Check that latency doesn’t drift upward over time. Use linear regression on the last 1–5 minutes of latency values to ensure slope ≈ 0.

from sklearn.linear\_model import LinearRegression

import numpy as np

X = np.arange(len(latencies)).reshape(-1, 1)

y = np.array(latencies)

slope = LinearRegression().fit(X, y).coef\_[0]

if abs(slope) < 0.5: # ~0.5ms/sec

latency\_is\_stable = True

* + - Set Initial Latency (L’): Once stable(L’ = average(p95\_latency over stable window). If our latency has noise (e.g., jumps between 110 and 150 ms), use p95 of a moving window instead of an average.
* Practical Parameters

| Parameter | Suggested Value | Description |
| --- | --- | --- |
| Start RPS | 1 | Initial low load |
| RPS Increment ΔR | +1 or +5 | Finer steps give more precise control |
| Wait Interval | 0 sec – 1 min | Wait before measuring impact |
| Latency Threshold (ΔL/ΔR) | α × L’ | Defines system instability slope |
| Max CPU Threshold | 75–80% | Avoid saturation |

* Metrics to Monitor During Each Step
  + p95\_latency – Use high-percentile (not average) latency to capture user-facing performance degradation.
  + ΔL/ΔR (Latency Gradient) – Detects sharp increases in latency as load increases.
  + CPU Usage (%) – Helps avoid resource starvation or throttling.
  + *(Optional)* GC Pause Time, Thread Queue Length – JVM-based services especially.
* Breaking Conditions — When to Stop Ramp-Up
  + Stop increasing load when:
    - Latency becomes sensitive to load

ΔR / ΔL ​=​ (Lcurrent​−Lprevious) / (Rcurrent​−Rprevious) ​​> Threshold

This means system is getting saturated or queuing is beginning to dominate response time.

* + - CPU usage exceeds safe threshold

CPUU​sage > Max threshold

This indicates approaching saturation or throttling (especially in Kubernetes).

1. **Backtrack to Last Stable Point**

Once a breaking condition is hit:

* Rollback to previous RPS (R₀)
* Record:
  + R₀ = max sustainable request rate
  + L₀ = corresponding p95 latency
  + cpu₀ = CPU usage at R₀
* This becomes your performance baseline

1. **Smart Stability Check**

To avoid false detection due to jitter or noisy metrics, apply statistical smoothing. Use Linear Regression:

* On a 5-minute sliding window of latency (p95\_latency(t))
* Fit latency vs time using least squares
* If slope ≈ 0, latency is stable

from sklearn.linear\_model import LinearRegression

window = last\_5\_minutes\_latency\_values

X = timestamps.reshape(-1, 1)

y = latencies

model = LinearRegression().fit(X, y)

slope = model.coef\_[0]

if abs(slope) < 0.5: # milliseconds per second

latency\_is\_stable = True

#### Why This Works:

* Balances real performance (latency) with infrastructure pressure (CPU).
* Avoids over/under-provisioning before scaling logic begins.
* Produces ground truth for later comparison during stress or optimization phases.

#### After This Phase:

We now have:

* A stable workload (R₀)
* A target latency (L₀)
* System pressure metrics (CPU₀, Memory₀)

### Stress Phase – Performance Deviation Analysis

After identifying the stable base request rate (R₀) and base latency (L₀) in the previous phase, the goal of this phase is to:

* Apply a higher workload (R₁ > R₀)
* Measure how the system responds in terms of latency and resource pressure
* Trigger vertical scaling if latency worsens
* Explore optimization if performance improves

Input from Previous Step:

* R₀ -> Base request rate (max sustainable rate without degradation)
* L₀ -> Corresponding base p95 latency
* Optional: cpu₀, mem₀ -> CPU and memory usage at baseline

**Increase the Request Rate**

Apply a higher request rate than the base: R₁ = R₀ + ΔR

We can choose ΔR as:

* +20% if you're testing moderate load increase
* +50–100% for aggressive surge testing

**Let System Stabilize**

Once R₁ is applied:

* Let the system run for a fixed time window (e.g., 1–3 minutes)
* This allows caching, CPU ramp-up, GC, and any retries to stabilize

**Measure Key Metrics**

Record:

* L₁ = p95 latency under R₁
* cpu₁ = CPU usage %
* mem₁ = Memory usage %
* Optional: GC time, queue depth, error rates

**Compare Against Baseline**

Case 1: Performance Degraded

if L₁ > L₀:

performance\_degraded = True

* Latency increased — system is struggling under R₁.
* This indicates:
  + CPU/memory bottlenecks
  + Saturated thread pools
  + GC pressure
  + DB/backend constraints

Case 2: Performance Improved or Same

if L₁ <= L₀:

performance\_stable\_or\_better = True

* Latency is stable or even lower than L₀ under higher load.
* This may mean:
  + The app is under-provisioned at R₀
  + JVM warmed up
  + Batching/pipelining became efficient
  + Thread pools now fully utilized

**Why Latency Can Drop under Higher Load**

* Low concurrency can keep workers idle.
* Higher load can improve thread pool utilization, cache hits, or batching.
* For ML inference, batching larger requests can reduce per-request overhead.
* Lower latency at higher RPS ≠ bug — may mean system is underloaded.

**What This Step Achieves**

* Pushes system toward peak performance
* Lets us dynamically decide whether to scale, optimize, or update baseline
* Prepares the system for realistic, fluctuating workloads

### Adaptive Vertical Scaling (with Bayesian Optimization)

Use Bayesian Optimization to smartly adjust CPU and memory requests for a Kubernetes pod to:

* Restore degraded performance (if L₁ > L₀)
* Improve performance further (if L₁ < L₀)
* Avoid overprovisioning or underutilization
* Minimize resource usage while keeping latency near or below L₀

**Why Bayesian Optimization?**

Unlike greedy step-wise increments, Bayesian Optimization:

* Explores intelligently based on past performance
* Balances exploration (try new CPU/mem combos) and exploitation (focus near best)
* Avoids wasteful trials and quickly converges to near-optimal allocations

**Inputs:**

From previous steps:

* R₀ = Base request rate
* R₁ = Applied request rate (R₁ > R₀)
* L₀ = Base latency
* L₁ = Current latency under stress

**System Design**

* Objective function:  
   Minimize latency + resource usage penalty
* Parameters to tune:
  + cpu\_request (e.g., from 200m to 1500m)
  + mem\_request (e.g., from 256Mi to 2Gi)

**Step-by-Step Execution**

1. **Define the Objective Function**

def objective(trial):

# Suggest CPU and memory request values

current\_cpu = get\_current\_cpu\_request() # e.g., 0.5 cores

current\_mem = get\_current\_memory\_request() # e.g., 1.0 Gi

cpu\_min = current\_cpu \* 0.5

mem\_min = current\_mem \* 0.5

cpu\_m = trial.suggest\_float("cpu", cpu\_min, cpu\_max)

mem\_g = trial.suggest\_float("memory", mem\_min, mem\_max)

# Apply the resource patch to the deployment

apply\_patch\_deployment(cpu=cpu\_m, mem=mem\_g)

# Wait for pod to stabilize

wait\_for\_stabilization()

# Measure current latency and resource usage

latency = measure\_p95\_latency()

cpu\_util = measure\_cpu\_usage\_pct()

mem\_util = measure\_memory\_usage\_pct()

# Normalize if usage is in 0–100%

if cpu\_util > 1: cpu\_util /= 100

if mem\_util > 1: mem\_util /= 100

# Compute performance gap

latency\_penalty = max(latency - initial\_latency, 0)

# CPU utilization penalty (dual-sided)

# Optimal CPU usage range: 60% to 85%

if cpu\_util < 0.6:

cpu\_penalty = (0.6 - cpu\_util) \* 10

elif cpu\_util > 0.85:

cpu\_penalty = (cpu\_util - 0.85) \* 15

else:

cpu\_penalty = 0

# Memory utilization penalty (dual-sided)

# Optimal Mem usage range: 60% to 85%

if mem\_util < 0.6:

mem\_penalty = (0.6 - mem\_util) \* 5

elif mem\_util > 0.85:

mem\_penalty = (mem\_util - 0.85) \* 10

else:

mem\_penalty = 0

# Final score: minimize this

score = latency\_penalty + cpu\_penalty + mem\_penalty

return score

1. Run the Optimizer (e.g., with Optuna)

Use Optuna’s study.stop() method from inside the objective function or a custom callback when:

* Latency reaches below a target
* Best score hasn't improved in N trials
* We detect instability

import optuna

study = optuna.create\_study(direction="minimize")

def early\_stopping\_callback(study, trial):

if study.best\_value is not None and study.best\_value < 0.01:

print("Early stopping: target performance achieved.")

study.stop()

study.optimize(objective, n\_trials=1000, callbacks=[early\_stopping\_callback])

This will:

* Try various CPU/Memory settings
* Observe their impact on latency and utilization
* Find the most balanced configuration

1. **Analyze and Apply Best Config**

best = study.best\_params

cpu\_best = best["cpu"]

mem\_best = best["memory"]

apply\_patch\_deployment(cpu=cpu\_best, mem=mem\_best)

**Condition-Based Control Logic**

Case A: If L₁ > L₀ (Performance Degraded)

* Use the above optimizer to scale up CPU/memory until latency improves.
* Use latency\_penalty to heavily prioritize regaining performance.

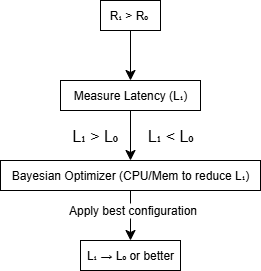
Case B: If L₁ < L₀ (Latency Improved)

* Use the optimizer to minimize latency further while reducing resources.
* Let the optimizer explore smaller CPU/mem values.
* Penalize waste (low utilization) to push resource efficiency.

**Benefits of This Approach**

| Feature | Greedy | Bayesian Optimization |
| --- | --- | --- |
| Finds global minimum | No | Yes |
| Avoids overprovisioning & underprovisioning | May be | Yess |
| Works under noisy data | No | Yes |
| Fast convergence | No | Yes (with few trials) |
| Easy to automate | Yes | Yes |

**Flow**

****

### Monitoring & Learning Loop (Reinforcement-Inspired Feedback System)

#### Goal

Create a self-adaptive system that continuously:

* Observes runtime performance
* Learns how different CPU/memory settings impact latency
* Builds a knowledge base of best configurations for various workloads
* Uses that knowledge to preemptively optimize for future workload changes

#### Key Concepts

| **Component** | **Description** |
| --- | --- |
| Latency gap (ΔL) | The difference between current latency and base latency (L₁ - L₀) |
| Resource headroom | The amount of unused CPU or memory (i.e., underutilized allocations) |
| Workload signature | Defined by request rate (RPS), time of day, or workload category |
| Knowledge base (KB) | A map of workload → best resource config (CPU/mem) |
| Feedback frequency | How often the loop executes (e.g., every 5 minutes) |

#### Step-by-Step Monitoring & Learning Loop

##### Every N Minutes (e.g., 5 Min): Collect Metrics

Use Prometheus

Collected metrics:

latency = measure\_p95\_latency()

cpu\_usage = measure\_cpu\_usage\_pct() # actual usage %

mem\_usage = measure\_memory\_usage\_pct()

cpu\_request = get\_requested\_cpu() # allocated request (cores)

mem\_request = get\_requested\_memory() # in Gi

rps = measure\_request\_rate()

##### Compute State Features

Define system performance state:

latency\_gap = latency - base\_latency # ΔL

cpu\_headroom = 1 - cpu\_usage # 0.3 means 30% unused

mem\_headroom = 1 - mem\_usage

utilization\_ratio = (cpu\_usage + mem\_usage) / 2

##### Evaluate Current Efficiency Score

cpu\_price\_ratio = get\_current\_cpu\_cost\_per\_core()

mem\_price\_ratio = get\_current\_memory\_cost\_per\_GB()

total\_price = cpu\_price\_ratio + mem\_price\_ratio

cpu\_penalty\_weight = cpu\_price\_ratio / total\_price

mem\_penalty\_weight = mem\_price\_ratio / total\_price

latency\_weight = 1.0 # always high

score = (

max(latency\_gap, 0) \* latency\_weight

+ cpu\_headroom \* cpu\_penalty\_weight

+ mem\_headroom \* mem\_penalty\_weight

)

Lower score = better system state

##### Define Workload Signature

Create a unique key to represent the workload context:

workload\_key = {

"rps": round(rps, 1),

"time\_window": get\_current\_time\_window(), # e.g., morning, peak, off-peak

"service": service\_name

}

##### Update Knowledge Base (KB)

Knowledge base stores mappings like:

{

"rps": 15.0,

"time\_window": "peak",

"service": "service-1"

} => {

"cpu": 0.6,

"memory": 1.25,

"score": 5.2

}

Update Logic:

if workload\_key not in knowledge\_base:

knowledge\_base[workload\_key] = current\_config

else:

if score < knowledge\_base[workload\_key]["score"]:

knowledge\_base[workload\_key] = current\_config # improved config

##### Action: Recommend or Apply Better Config

On next loop or during autoscaling:

if workload\_key in knowledge\_base:

best\_config = knowledge\_base[workload\_key]

if current\_config != best\_config:

apply\_patch\_deployment(cpu=best\_config["cpu"], mem=best\_config["memory"])

This enables the system to self-optimize proactively, not reactively.

#### Optional: Model the Latency-Resource Surface

As data accumulates, fit a predictive model:

latency = f(cpu, memory, rps) + ε

Use:

* Regression Trees
* Gaussian Process
* Neural Network
* Bayesian Surface Estimation

This enables:

* Predicting what CPU/mem combo will keep latency near L₀
* Simulating impact before deploying change

#### System Architecture Flow

[Metrics Collection] -> [Compute Score & Headroom] -> [Identify Workload Signature (RPS, Time)] -> [Check & Update Knowledge Base] -> [Recommend / Apply Better Config] -> [Wait N Minutes & Repeat]

| **Feature** | **Benefit** |
| --- | --- |
| Continuous learning | Adapts to workload shifts |
| No manual tuning | Builds optimal resource map over time |
| Avoids overprovisioning | Detects and corrects waste automatically |
| Preemptive autoscaling | Applies best config before degradation begins |

### Optional: How to Connect Bayesian Surface Estimation to Our System

#### Step 1: Collect Training Data

From each Monitoring Loop iteration, store:

X = [cpu\_request, mem\_request, rps]

y = [observed\_latency]

Build a dataset

#### Step 2: Train a Bayesian Regression Model

Gaussian Process Regression (GPR):

from sklearn.gaussian\_process import GaussianProcessRegressor

from sklearn.gaussian\_process.kernels import RBF, WhiteKernel

import numpy as np

# X: [cpu, memory, rps]

# y: latency

X = np.array(training\_data\_inputs) # shape: (n\_samples, 3)

y = np.array(training\_data\_outputs) # shape: (n\_samples,)

kernel = RBF(length\_scale=1.0) + WhiteKernel(noise\_level=1)

model = GaussianProcessRegressor(kernel=kernel)

model.fit(X, y)

This model now estimates latency = f(cpu, memory, rps) ± uncertainty

#### 

#### Step 3: Predict Latency for Candidate Configurations

# Predict latency at 0.6 cores, 1.2 Gi, and 14 RPS

latency\_pred, std = model.predict([[0.6, 1.2, 14]], return\_std=True)

print(f"Expected latency: {latency\_pred[0]:.2f} ms ± {std[0]:.2f}")

#### Step 4: Use Predictions in the Feedback Loop

Before deciding to apply a config:

1. Generate several candidate configurations near the current point
2. Predict latency using GPR
3. Choose the one with:
   * lowest latency
   * minimal CPU/memory
   * within a confidence range (e.g., std < 20 ms)

candidates = [

[0.5, 1.0, rps],

[0.6, 1.0, rps],

[0.7, 1.2, rps],

...

]

best\_score = float("inf")

best\_config = None

for config in candidates:

pred, std = model.predict([config], return\_std=True)

if std[0] < 15: # avoid high uncertainty

score = pred[0] + 5 \* config[0] + 3 \* config[1] # latency + weighted CPU/mem cost

if score < best\_score:

best\_score = score

best\_config = config

# Apply best\_config: [cpu, memory, rps]

apply\_patch\_deployment(cpu=best\_config[0], mem=best\_config[1])

#### Step 5: Continually Retrain the Model

Every N iterations (e.g., every 10 samples), retrain:

if len(training\_data\_inputs) % 10 == 0:

model.fit(X, y)

This keeps the surface estimation up-to-date as load patterns change.

| **Benefit** | **Description** |
| --- | --- |
| Predicts before deploying | Reduces unnecessary patching and rolling restarts |
| Learns a generalized latency surface | Not just for one RPS — works across any workload |
| Handles uncertainty | Uses ± std to skip bad recommendations |
| Integrates seamlessly | Fits directly into your Step 5 feedback loop |

#### Tools We Can Use

| **Tool** | **Purpose** |
| --- | --- |
| scikit-learn | GPR, Bayesian Ridge |
| GPy, GPyTorch | More scalable GPs |
| Optuna | Bayesian optimizer with study.sampler |
| BayesianOptimization package | High-level interface to fit surfaces |

#### Architecture Update (with Bayesian Layer)

[Metrics Collector] -> [Training Data (X, y) Store] -> [Bayesian Surface Estimator (GPR)] -> [Predict latency for candidate configs] -> [Choose optimal (CPU, mem) combo] -> [Patch Deployment + Feedback]