Serverless Function Runtime: Design Document

Overview

This system provides a platform for running ephemeral, event-driven functions without managing servers. The key architectural challenge it solves is balancing strong isolation and fast startup times (cold start optimization) while efficiently scaling from zero to handle unpredictable workloads.

> This guide is meant to help you understand the big picture before diving into each milestone. Refer back to it whenever you need context on how components connect.

Context and Problem Statement

> **Milestone(s):** This foundational section establishes the core problem space that all subsequent milestones address: balancing isolation, speed, and efficiency in serverless function execution.

Serverless computing represents a paradigm shift in how applications are built and deployed. Instead of provisioning and managing long-lived servers, developers write discrete functions that automatically scale and execute in response to events. The platform handles all infrastructure concerns—provisioning, scaling, patching, and monitoring—allowing developers to focus solely on business logic. This model offers compelling advantages: **cost efficiency** (pay only for execution time), **elastic scaling** (automatic response to load), and **reduced operational overhead** (no server management).

However, beneath this simplified abstraction lies a complex technical challenge. Functions must execute with strong **isolation** (preventing one tenant's code from affecting another) while maintaining **fast startup times** (responding to requests within milliseconds) and **resource efficiency** (minimizing overhead for idle functions). These requirements often conflict, creating the central tension in serverless runtime design.

Mental Model: The Instant Coffee Machine vs. The Barista

To understand the fundamental trade-offs in serverless computing, consider two approaches to serving coffee:

The Instant Coffee Machine (Fast but Limited)

- **Operation**: Press a button → get coffee instantly
- **Preparation**: Coffee is pre-ground, pre-measured, and pre-packaged in single-serving capsules
- **Customization**: Limited to pre-defined options (espresso, lungo, americano)
- **Isolation**: Each capsule is completely self-contained; no cross-contamination between servings
- **Cleanup**: Capsule is discarded; machine is ready for next user immediately
- **Analogous to**: Pre-initialized, templated execution environments with limited runtime customization

The Barista (Customizable but Slower)

- **Operation**: Place order → barista grinds beans, tamps, steams milk, pours art
- **Preparation**: Everything done fresh per order with high-quality ingredients
- **Customization**: Infinite variations (bean origin, grind size, milk temperature, syrup)
- **Isolation**: Shared equipment cleaned between uses (risk of cross-contamination)
- **Cleanup**: Thorough cleaning of portafilter, steam wand, and work area required
- **Analogous to**: Full container or VM per function with complete environment customization

The instant coffee machine represents the ideal of serverless: **predictable, fast, and isolated execution**. Each function invocation gets its own "capsule"—a pre-packaged execution environment that starts instantly. However, this comes at the cost of flexibility: you're limited to the runtimes and dependencies that fit in the capsule format.

The barista model offers maximum flexibility: you can bring any coffee beans (dependencies), any milk (runtime version), and request any preparation method (custom initialization). But each order takes time to prepare, and thorough cleaning is needed between customers to prevent flavor mixing (security isolation).

The challenge in building a serverless runtime is creating a system that feels like an **instant coffee machine** to the user (fast, simple, predictable) while internally supporting the **flexibility of a barista** (arbitrary code, dependencies, runtimes). We need the speed of pre-packaged execution with the isolation guarantees of fresh preparation for each customer.

The Core Technical Challenge: Isolation vs. Speed

The fundamental tension in serverless runtime design manifests as a three-way optimization problem between **isolation strength**, **startup latency**, and **resource efficiency**.

Isolation Requirements Serverless platforms typically host code from multiple untrusted tenants on shared hardware. Strong isolation is non-negotiable to prevent:

- **Resource interference**: One function consuming all CPU/memory/IO, starving others
- **Security breaches**: A function accessing another tenant's data or code
- **Noisy neighbors**: Performance degradation due to shared resources
- **Privilege escalation**: Gaining host-level access through kernel vulnerabilities

Startup Latency Constraints The "serverless promise" includes sub-second response times, even for the first invocation (cold start). Real-world applications have strict latency budgets:

- **User-facing APIs**: Typically require < 200ms total response time
- **Event processing**: Often batch processing with < 1s latency requirements
- **Cold start penalty**: The additional time to initialize a function from scratch

Resource Efficiency Demands Since customers pay only for execution time, the platform must minimize overhead:

- **Memory footprint**: Idle environments consume memory even when not executing
- **CPU overhead**: Environment creation/teardown consumes CPU cycles
- **Storage costs**: Function packages and runtime images require storage
- **Network bandwidth**: Downloading dependencies on each cold start

This creates a challenging optimization landscape:

Goal	Improves	Conflicts With
Stronger isolation	Security, reliability	Startup speed (more setup), Resource efficiency (more overhead)
Faster startup	User experience, throughput	Isolation (less validation), Efficiency (pre-warming costs)
Higher efficiency	Cost reduction, density	Isolation (sharing resources), Startup speed (less pre-warming)

> **Key Insight**: The "cold start problem" is not just about speed—it's the visible symptom of this deeper trade-off. When we optimize for isolation (fresh environment per invocation) and efficiency (no pre-warmed instances), we necessarily sacrifice startup speed. Conversely, when we optimize for speed (pre-warmed pools), we sacrifice efficiency (idle resource consumption) and potentially isolation (shared/reused environments).

Formalizing the Problem Given:

- A set of functions $F = \{f_1, f_2, ..., f_n\}$ with arbitrary code and dependencies
- A stream of invocation requests $R = \{r_1(t_1), r_2(t_2), ...\}$ at unpredictable times
- Hardware resources $H = \{CPU, memory, storage, network\}$ shared across tenants
- Security requirements $S = \{isolation, quotas, audit\}$ for multi-tenancy

Design a runtime system that:

1. **Minimizes P99 latency**: $P99(response\_time(r_i)) \leq L\_max$ (e.g., 200ms)
2. **Enforces strong isolation**: $\forall f_i, f_j \in F, i \neq j: isolation(f_i, f_j) \geq I\_min$
3. **Maximizes resource utilization**: $utilization(H) \to 1$ while $cost \to min$
4. **Scales elastically**: $instances(f_i) \propto load(f_i)$ with minimal hysteresis

This is the core problem our serverless runtime must solve, and each milestone represents a piece of the solution.

Survey of Existing Approaches

The industry has explored multiple isolation technologies for serverless workloads, each with different trade-offs on the isolation-speed-efficiency spectrum. Understanding these approaches informs our architecture decisions.

1. Linux Container Namespaces & cgroups (Docker-like)

Aspect	Characteristics
Isolation Mechanism	Kernel namespaces (pid, net, mount, user, ipc, uts) + cgroups for resource limits
Startup Time	100-500ms for fresh container, ~50ms for pre-initialized
Isolation Strength	Moderate (shared kernel, potential for kernel exploits)
Memory Overhead	Low (~10MB per container for runtime)
Implementation Complexity	Medium (leveraging existing container runtimes)
Multi-tenant Security	Requires additional hardening (seccomp, AppArmor, SELinux)

Pros:

- Mature ecosystem (Docker, containerd, CRI-O)
- Fast startup with image layering and copy-on-write
- Low overhead compared to full virtualization
- Excellent tooling for packaging and distribution

Cons:

- Shared kernel creates larger attack surface
- Requires careful configuration for production multi-tenancy
- Namespace escape vulnerabilities periodically discovered
- Less familiar to non-Linux developers

Use Cases: AWS Lambda (2018-2020), Google Cloud Run, OpenFaaS, most on-prem serverless platforms.

2. MicroVMs (Firecracker)

Aspect	Characteristics
Isolation Mechanism	Hardware virtualization (KVM) with minimal, specialized hypervisor
Startup Time	125-150ms for fresh microVM, <10ms from snapshot
Isolation Strength	Very high (hardware-enforced isolation)
Memory Overhead	Moderate (~5MB per microVM + guest memory)
Implementation Complexity	High (managing VMM lifecycle, device models)
Multi-tenant Security	Excellent (hardware isolation, minimal attack surface)

Pros:

- Near-bare-metal performance with hardware isolation
- Extremely fast snapshot/restore (<10ms)
- Minimal attack surface (Firecracker <50K lines vs QEMU 1.5M)
- Designed specifically for serverless workloads

Cons:

- Still heavier than containers
- More complex to integrate with container ecosystem
- Limited device support (intentionally)
- Requires virtualization extensions (Intel VT-x/AMD-V)

Use Cases: AWS Lambda (current), AWS Fargate, Cloudflare Workers (isolates).

3. Process-based Sandboxes (gVisor, nsjail)

Aspect	Characteristics
Isolation Mechanism	Userspace kernel implementation + namespace isolation
Startup Time	20-100ms (lightweight process creation)
Isolation Strength	High (syscall filtering, userspace kernel)
Memory Overhead	Low to moderate (shared runsc runtime)
Implementation Complexity	High (custom kernel implementation)
Multi-tenant Security	Very good (syscall interception, no shared kernel)

Pros:

- No virtualization required
- Fast startup (lightweight processes)
- Strong isolation without VM overhead
- Compatible with container images

Cons:

- Performance overhead for syscall-heavy workloads
- Compatibility issues with some syscalls
- Complex implementation and debugging
- Less battle-tested than containers/VMs

Use Cases: Google Cloud Run (gVisor), sandboxed CI/CD environments.

4. Language-Specific Runtimes (WebAssembly)

Aspect	Characteristics
Isolation Mechanism	Capability-based security within runtime
Startup Time	<1ms (pre-compiled bytecode)
Isolation Strength	Language-dependent (memory-safe languages higher)
Memory Overhead	Very low (shared runtime, per-instance memory)
Implementation Complexity	High (per-language runtime modifications)
Multi-tenant Security	Good within runtime, depends on host isolation

Pros:

- Extremely fast startup (pre-compiled)
- Portable across platforms
- Fine-grained resource control
- Growing ecosystem

Cons:

- Limited language support
- System call interface still evolving
- Less mature tooling
- Runtime-specific vulnerabilities possible

Use Cases: Fastly Compute@Edge, Fermyon Spin, experimental platforms.

Comparison Table

Technology	Startup Time	Isolation	Overhead	Maturity	Our Use Case
Containers	100-500ms	Moderate	Low	Very High	Baseline for comparison, good starting point
MicroVMs	10-150ms	Very High	Moderate	High	Production multi-tenant with strong security
Process Sandboxes	20-100ms	High	Low-Moderate	Medium	When virtualization unavailable, good security
WebAssembly	<1ms	Language-dependent	Very Low	Emerging	Specialized use cases, fastest cold starts

Decision: Container-Based Isolation for Initial Implementation

- **Context**: We need a practical starting point that balances implementation complexity, performance, and isolation while leveraging existing ecosystems.
- **Options Considered**:
  1. Linux containers (namespaces + cgroups)
  2. Firecracker microVMs
  3. gVisor process sandboxes
- **Decision**: Start with container-based isolation using runc/containerd.
- **Rationale**:
  - Containers provide sufficient isolation for many use cases with proper hardening
  - Vast ecosystem for packaging, distribution, and management
  - Easier debugging and development experience
  - Can be augmented with additional security (seccomp, AppArmor)
  - Natural migration path to microVMs later via containerd's shim API
- **Consequences**:
  - We'll need to implement additional security hardening for production
  - Cold starts will be slower than microVM snapshot/restore
  - We can later swap container runtime for Firecracker with minimal API changes

Industry Trend: Blending Approaches Modern serverless platforms often combine multiple isolation technologies:

- **AWS Lambda**: Uses Firecracker microVMs but maintains container compatibility via OCI images
- **Google Cloud Run**: Offers both gVisor and container-based isolation options
- **Azure Container Instances**: Uses Hyper-V isolation for multi-tenant, containers for single-tenant
- **OpenFaaS**: Primarily container-based but supports Kata Containers (lightweight VMs)

The trend is toward **defense-in-depth**: using stronger isolation for less-trusted workloads while optimizing for speed where appropriate. Our architecture should support this evolution.

Common Pitfalls in Isolation Technology Selection

⚡ **Pitfall: Over-engineering isolation for all use cases**

- **Description**: Implementing the strongest possible isolation (e.g., full VMs) for all functions, even internal trusted functions.
- **Why it's wrong**: Creates unnecessary overhead, slower startups, and higher costs without corresponding security benefit.
- **How to avoid**: Implement tiered isolation levels based on function trust level or tenant requirements.

⚡ **Pitfall: Underestimating kernel attack surface**

- **Description**: Assuming container isolation is "secure enough" without additional hardening.
- **Why it's wrong**: New container escape vulnerabilities are regularly discovered (e.g., CVE-2019-5736 runc escape).
- **How to avoid**: Always enable seccomp, AppArmor/SELinux, and capability dropping. Consider gVisor or microVMs for untrusted code.

⚡ **Pitfall: Ignoring initialization time variability**

- **Description**: Assuming all functions have similar startup characteristics regardless of runtime.
- **Why it's wrong**: JVM functions take ~1s+ to start, Python ~100ms, Go binary ~10ms.
- **How to avoid**: Measure actual startup times per runtime and adjust warm pool strategies accordingly.

⚡ **Pitfall: Tight coupling to specific isolation technology**

- **Description**: Building the entire system around Docker API calls or Firecracker-specific features.
- **Why it's wrong**: Makes migration to new technologies difficult as the ecosystem evolves.
- **How to avoid**: Define abstraction interfaces (ContainerManager, SandboxManager) that can have multiple implementations.

Implementation Guidance

While this section focuses on problem definition rather than implementation, we'll establish the foundational project structure and provide tools to explore the isolation technologies discussed.

A. Technology Recommendations Table

Component	Simple Option (Learning)	Production Option (Advanced)
Isolation	Docker Engine API	containerd + runc
Orchestration	Custom Go controller	Kubernetes Custom Resources
Storage	Local filesystem	S3-compatible object storage
Networking	Docker bridge network	CNI plugins (bridge, macvlan)
Monitoring	Prometheus metrics	OpenTelemetry + distributed tracing

B. Recommended File/Module Structure

```
serverless-runtime/
├── cmd/
│   ├── controller/          # Main orchestrator
│   │   └── main.go
│   ├── gateway/             # HTTP gateway
│   │   └── main.go
│   └── worker/              # Worker node daemon
│       └── main.go
├── internal/
│   ├── api/                 # Public API definitions
│   │   ├── types.go         # Request/response types
│   │   └── server.go        # HTTP handlers
│   ├── container/           # Container abstraction layer
│   │   ├── manager.go       # ContainerManager interface
│   │   ├── docker.go        # Docker implementation
│   │   └── firecracker.go   # Firecracker implementation (future)
│   ├── registry/            # Function package storage
│   │   ├── storage.go       # Storage interface
│   │   ├── s3.go            # S3 implementation
│   │   └── local.go         # Local filesystem implementation
│   ├── scheduler/           # Request routing & scaling
│   │   ├── gateway.go       # HTTP gateway logic
│   │   ├── scaler.go        # Auto-scaling logic
│   │   └── queue.go         # Request queue
│   └── runtime/             # Language-specific runtime support
│       ├── builder.go       # Function package builder
│       ├── go.go            # Go runtime
│       ├── python.go        # Python runtime
│       └── nodejs.go        # Node.js runtime
├── pkg/
│   ├── function/            # Function definition types
│   │   └── types.go
│   ├── invocation/          # Invocation tracking
│   │   └── types.go
│   └── metrics/             # Metrics collection
│       └── collector.go
├── scripts/
│   ├── benchmark/           # Performance testing
│   └── setup/               # Environment setup
├── go.mod
└── README.md
```

C. Infrastructure Starter Code: Container Exploration Tool

To understand the practical differences between isolation technologies, here's a tool that measures startup times:

```go
// cmd/explore/main.go - Container isolation exploration tool

package main


import (

	"context"

	"fmt"

	"log"

	"time"


	"github.com/docker/docker/api/types"

	"github.com/docker/docker/api/types/container"

	"github.com/docker/docker/client"

)


type IsolationBenchmark struct {

	Technology string

	Image      string

	Command    []string

}


func main() {

	benchmarks := []IsolationBenchmark{

		{

			Technology: "Container (Alpine)",

			Image:      "alpine:latest",

			Command:    []string{"echo", "Hello from container"},

		},

		{

			Technology: "Container (Python)",

			Image:      "python:3.9-slim",

			Command:    []string{"python", "-c", "print('Python ready')"},

		},

		// Add more benchmarks for different technologies

	}


	ctx := context.Background()

	cli, err := client.NewClientWithOpts(client.FromEnv, client.WithAPIVersionNegotiation())

	if err != nil {

		log.Fatal(err)

	}


	fmt.Println("Isolation Technology Benchmark")

	fmt.Println("==============================")
```

```
	for _, bench := range benchmarks {
		fmt.Printf("\nBenchmarking: %s\n", bench.Technology)

		// Pull image (simulates dependency download)
		start := time.Now()
		fmt.Printf("  Pulling image %s... ", bench.Image)
		reader, err := cli.ImagePull(ctx, bench.Image, types.ImagePullOptions{})
		if err != nil {
			log.Printf("Failed to pull image: %v", err)
			continue
		}
		// Drain the reader
		buf := make([]byte, 1024)
		for {
			_, err := reader.Read(buf)
			if err != nil {
				break
			}
		}
		reader.Close()
		pullTime := time.Since(start)
		fmt.Printf("%v\n", pullTime)

		// Create container (environment setup)
		start = time.Now()
		fmt.Printf("  Creating container... ")
		resp, err := cli.ContainerCreate(ctx, &container.Config{
			Image: bench.Image,
			Cmd:   bench.Command,
		}, nil, nil, nil, "")
		if err != nil {
			log.Printf("Failed to create container: %v", err)
			continue
		}
		createTime := time.Since(start)
		fmt.Printf("%v\n", createTime)

		// Start container (runtime initialization)
		start = time.Now()
		fmt.Printf("  Starting container... ")
		err = cli.ContainerStart(ctx, resp.ID, types.ContainerStartOptions{})
```

```
                if err != nil {
                        log.Printf("Failed to start container: %v", err)
                        continue
                }
                startTime := time.Since(start)
                fmt.Printf("%v\n", startTime)

                // Wait for completion (execution time)
                statusCh, errCh := cli.ContainerWait(ctx, resp.ID, container.WaitConditionNotRunning)
                select {
                case err := <-errCh:
                        if err != nil {
                                log.Printf("Error waiting for container: %v", err)
                        }
                case <-statusCh:
                        // Container finished
                }

                // Clean up
                _ = cli.ContainerRemove(ctx, resp.ID, types.ContainerRemoveOptions{})

                fmt.Printf("  Total cold start: %v\n", pullTime+createTime+startTime)
        }
}
```

D. Core Logic Skeleton: Isolation Interface

```go
// internal/container/manager.go - Abstract isolation interface

package container


import (

        "context"

        "io"

        "time"

)


// ContainerManager defines the interface for managing isolated execution environments.

// This abstraction allows swapping Docker for Firecracker or other technologies.

type ContainerManager interface {

        // CreateContainer creates a new isolated environment from an image

        // TODO 1: Validate image exists locally, pull if missing with retry logic

        // TODO 2: Generate unique container ID for tracking

        // TODO 3: Apply resource limits (CPU, memory) from config

        // TODO 4: Set up namespaces and cgroups for isolation

        // TODO 5: Mount function code volume at /function

        // TODO 6: Configure network namespace (bridge, host, or none)

        // TODO 7: Set up seccomp and AppArmor profiles for security

        CreateContainer(ctx context.Context, req CreateContainerRequest) (*Container, error)



        // StartContainer begins execution of the container

        // TODO 1: Validate container is in created state

        // TODO 2: Start the container process with proper stdio pipes

        // TODO 3: Wait for container to reach running state (health check)

        // TODO 4: Start timeout timer for startup

        // TODO 5: Log container startup events for debugging

        StartContainer(ctx context.Context, containerID string) error



        // ExecuteInContainer runs a command inside a running container

        // TODO 1: Validate container is running

        // TODO 2: Create exec instance with command and environment

        // TODO 3: Attach to stdin/stdout/stderr streams

        // TODO 4: Start execution and monitor for completion

        // TODO 5: Handle timeouts and cancellations

        // TODO 6: Capture exit code and output

        ExecuteInContainer(ctx context.Context, containerID string, cmd []string,

                input io.Reader, output io.Writer) (int, error)



        // StopContainer gracefully stops a running container
```

```
        // TODO 1: Send SIGTERM to container process
        // TODO 2: Wait for graceful shutdown (configurable timeout)
        // TODO 3: If timeout exceeded, send SIGKILL
        // TODO 4: Ensure all resources are cleaned up
        StopContainer(ctx context.Context, containerID string, timeout time.Duration) error

        // RemoveContainer cleans up container resources
        // TODO 1: Ensure container is stopped
        // TODO 2: Remove container filesystem
        // TODO 3: Clean up network namespace
        // TODO 4: Remove cgroup
        // TODO 5: Delete container metadata
        RemoveContainer(ctx context.Context, containerID string) error

        // GetContainerStats returns resource usage statistics
        // TODO 1: Read cgroup metrics for CPU, memory, I/O
        // TODO 2: Calculate utilization percentages
        // TODO 3: Include network statistics if available
        // TODO 4: Return structured metrics for monitoring
        GetContainerStats(ctx context.Context, containerID string) (*ContainerStats, error)

        // ListContainers returns all managed containers
        // TODO 1: Query container runtime for all containers
        // TODO 2: Filter by labels (e.g., function-id)
        // TODO 3: Include status and resource information
        // TODO 4: Support pagination for large numbers
        ListContainers(ctx context.Context, filter ListFilter) ([]Container, error)
}

// Container represents an isolated execution environment
type Container struct {
        ID        string
        Image     string
        Status    ContainerStatus
        CreatedAt time.Time
        Labels    map[string]string
        // TODO: Add resource limits, network info, mount points
}

type ContainerStatus string

const (
```

```
        StatusCreated    ContainerStatus = "created"
        StatusRunning    ContainerStatus = "running"
        StatusPaused     ContainerStatus = "paused"
        StatusRestarting ContainerStatus = "restarting"
        StatusExited     ContainerStatus = "exited"
        StatusDead       ContainerStatus = "dead"
)

type CreateContainerRequest struct {
        Image        string
        Cmd          []string
        Env          []string
        Labels       map[string]string
        CPUQuota     int64    // in microseconds
        MemoryLimit  int64    // in bytes
        NetworkMode  string
        Binds        []string // volume mounts
        // TODO: Add security options, working directory, user
}

// internal/container/docker.go - Docker implementation skeleton
type DockerManager struct {
        client *docker.Client
}

func NewDockerManager() (*DockerManager, error) {
        // TODO 1: Initialize Docker client with proper options
        // TODO 2: Verify Docker daemon is reachable
        // TODO 3: Set up context with timeout
        // TODO 4: Configure client for API version negotiation
        return nil, nil
}

func (dm *DockerManager) CreateContainer(ctx context.Context, req CreateContainerRequest) (*Container, error) {
        // TODO: Implement using Docker SDK
        return nil, nil
}
```

E. Language-Specific Hints for Go

1. **Docker SDK**: Use `github.com/docker/docker/client` for the official Go client. Remember to negotiate API version: `client.WithAPIVersionNegotiation()`.
2. **Context Usage**: Always pass `context.Context` to container operations for cancellation and timeouts. Use `context.WithTimeout` for operations that should have deadlines.
3. **Resource Limits**: When setting cgroup limits via Docker:

```go
// CPU: 100000 means 100% of one CPU core, 50000 means 50%
HostConfig: &container.HostConfig{
    Resources: container.Resources{
        CPUQuota:  50000,  // 0.5 CPU
        Memory:    128 * 1024 * 1024,  // 128MB
    },
}
```

4. **Error Handling**: Container operations can fail in many ways. Check for:
   - `errdefs.IsNotFound(err)` for missing images/containers
   - `errdefs.IsConflict(err)` for concurrent modifications
   - `errdefs.IsNotModified(err)` for idempotent operations
5. **Cleanup**: Always implement defer or finally blocks to clean up containers, especially in error paths to prevent resource leaks.

F. Milestone Checkpoint: Understanding Isolation Technologies

After studying this section, you should be able to:

1. **Run the exploration tool** to measure container startup times:

```bash
cd serverless-runtime
go run cmd/explore/main.go
```

Expected output shows timing for different container types.

2. **Understand the trade-offs** by answering:
   - Why would you choose containers over VMs for internal functions?
   - What additional security measures would you add to containers for production?
   - How does startup time change with image size?
3. **Experiment with security profiles**:

```bash
# Run a container with seccomp profile
docker run --security-opt seccomp=default.json alpine echo "hello"

# Run without any capabilities
docker run --cap-drop=ALL alpine echo "hello"
```

G. Debugging Tips: Container Startup Issues

Symptom	Likely Cause	How to Diagnose	Fix
Container fails to start with "exec format error"	Wrong architecture or corrupted image	`docker image inspect <image>` check architecture	Pull correct image, verify download integrity
Container starts but immediately exits	Missing command or entrypoint	`docker logs <container>` for error messages	Specify CMD in Dockerfile or override at runtime
Permission denied errors in container	User/group mismatch or SELinux/AppArmor	`docker run --security-opt apparmor=unconfined` test	Fix file permissions or adjust security profile
High memory usage reported	Memory limit not applied	`docker stats` shows actual usage	Verify memory limit in container config, check for memory leaks
Slow container startup	Large image layers or slow storage	`time docker run --rm alpine true` measure	Use smaller base images, enable layer caching

Goals and Non-Goals

> **Milestone(s):** This foundational section establishes the guiding principles and constraints that shape all subsequent design decisions across Milestones 1-5. It defines the measurable success criteria for the entire runtime system and explicitly scopes the project to a manageable, focused initial implementation.

This section crystallizes the **what** and **what not** of our serverless runtime. Given the vast landscape of possible features in cloud computing, explicit boundaries are essential to maintain project focus, allocate engineering effort effectively, and set clear expectations. The goals are derived from the core value proposition of serverless computing: effortless scaling, cost efficiency, and developer productivity. The non-goals represent deliberate omissions—features that, while potentially valuable, would derail us from our primary mission or are better handled by adjacent systems.

Guiding Principles

Before enumerating specific goals, we establish three overarching principles that inform every architectural choice:

1. **Isolation is Non-Negotiable:** In a multi-tenant environment, one function must not be able to observe, interfere with, or exhaust resources allocated to another. Security and fairness are foundational.
2. **Cold Start Latency is the Primary Performance Metric:** The time from invocation request to function code beginning execution defines user-perceived performance. Optimizing this is our central technical challenge.
3. **Scale-to-Zero is a Core Efficiency Requirement:** The runtime must truly be serverless, consuming no compute resources when there is no demand, while balancing this with the cold start penalty.

Goals

The following table defines the specific, measurable outcomes that constitute a successful implementation of our serverless function runtime. Each goal is tied to a key user benefit and has concrete, testable acceptance criteria.

Goal	Success Criteria	Rationale & User Benefit
G1: Strong Multi-Tenant Isolation	1. A function cannot access another function's memory, files, or network traffic. 2. A function cannot cause a denial-of-service via resource exhaustion (CPU, memory, disk, PIDs) to the host or other functions. 3. Functions run with no inherent privileges; they cannot escalate to host root or modify host kernel parameters.	Provides security and predictability. Users can run untrusted or sensitive code without fear of cross-contamination or hostile actors. This is the bedrock of a public or multi-team platform.
G2: Sub-Second Cold Start Latency (P95)	The 95th percentile latency for a cold start invocation (including environment provisioning, code loading, and runtime initialization) is under 1 second for supported runtimes (Go, Python, Node.js).	Delivers a responsive user experience. Functions feel "instant" even after periods of inactivity, making serverless viable for user-facing, synchronous APIs.
G3: Efficient Scale-to-Zero	1. The system releases all compute resources (CPU, memory) for a function after a configurable idle period (e.g., 5-15 minutes). 2. The cost of maintaining zero active instances is negligible (only metadata storage).	Enables true pay-per-use pricing. Users incur no cost when their code isn't running, which is the primary economic appeal of serverless.
G4: Rapid, Metric-Driven Autoscaling	1. The system can scale from zero to *N* concurrent instances to match incoming request load within seconds. 2. Scaling decisions are based on observable metrics (requests-per-second, concurrent executions) with configurable thresholds. 3. Scaling oscillations (thrashing) are minimized through cooldown periods and hysteresis.	Handles unpredictable or bursty workloads automatically. Users do not need to provision or manage capacity; the platform seamlessly absorbs traffic spikes.
G5: Polyglot Runtime Support	The platform supports at least three popular language runtimes: Go (compiled binary), Python (interpreted), and Node.js (interpreted). Each is provided with a standard library environment and a mechanism for user-supplied dependencies.	Meets developers where they are. Teams can use their preferred language without being locked into a proprietary ecosystem, increasing adoption.
G6: Simple, Declarative Function Packaging	Developers can deploy a function by providing source code and a simple declaration file (e.g., `func.yaml`) specifying the handler, runtime, and dependencies. The system handles the rest: building, bundling, and storage.	Reduces operational complexity and cognitive load. The developer experience is focused on business logic, not infrastructure plumbing.
G7: Comprehensive Observability	The system emits structured logs, metrics (invocations, durations, errors, cold starts), and traces for every function invocation. This data is accessible via an API or integrated dashboard.	Enables debugging, performance tuning, and cost analysis. Users gain visibility into their application's behavior without manual instrumentation.
G8: Predictable, Enforceable Resource Limits	Every function invocation is executed within strictly enforced limits for execution time (timeout), memory, and CPU allocation. Limits are defined at deployment and cannot be exceeded.	Prevents buggy or malicious code from consuming unbounded resources. Provides cost and performance predictability for platform operators and users.

Non-Goals

Equally important are the features we explicitly **will not** build in the initial version. These non-goals prevent scope creep and allow us to ship a robust, focused V1. Each is listed with a reason for exclusion, which is often that the feature is out of scope, overly complex, or better provided by a complementary system.

Non-Goal	Reason for Exclusion	Potential Future Consideration
NG1: Stateful Function Instances	The runtime assumes functions are stateless. We will not provide durable, mutable local disk or memory that persists across invocations.	State belongs in managed databases, caches, or object storage. Adding instance-persistent state dramatically complicates scaling, recovery, and data durability.
NG2: Long-Running Functions (Hours/Days)	The system is optimized for short, ephemeral execution (seconds to minutes). We will not support functions that run for hours or days.	Use a traditional container service or virtual machine for long-running processes. Our scheduling, billing, and fault recovery models are built for brevity.
NG3: Custom Hardware or GPU Access	Functions cannot request or access specialized hardware like GPUs, TPUs, or high-performance network interfaces.	The isolation and scheduling complexity for heterogeneous hardware is immense. This is a specialized niche outside our core focus on general-purpose compute.
NG4: Complex Event Source Integration	We will not build first-class, built-in integrations with dozens of event sources (e.g., Kafka, RabbitMQ, DynamoDB streams). Our primary trigger is HTTP.	The function runtime is an execution layer. Event sources should be configured externally (e.g., via a message queue that calls our HTTP endpoint). This keeps the runtime simple and composable.
NG5: Advanced Networking (VPC Peering, Private Endpoints)	Functions initially run in a shared, platform-managed network with outbound internet access. We will not support complex networking topologies like VPC peering or placement in a user's private subnet.	Networking is a complex domain that conflicts with our goal of simplicity and multi-tenancy. This can be added later if enterprise demand is high, but it significantly increases operational burden.
NG6: Function Composition or Workflow Orchestration	We will not provide a built-in DSL or engine for chaining functions together into workflows (e.g., parallel execution, fan-out/fan-in).	Workflow orchestration is a separate concern. Our runtime executes a single unit of work. Orchestration can be implemented at the application layer or by a dedicated service (like AWS Step Functions) that invokes our functions.
NG7: Real-Time Live Code Updates	A deployed function version is immutable. We will not support hot-swapping code or configuration into a running function instance.	Immutability simplifies versioning, rollbacks, and consistency. Updates require a new deployment, which creates a new immutable version. This is a core principle for reliability.
NG8: Full Operating System Compatibility	The execution environment is a constrained, purpose-built runtime, not a full general-purpose OS. We will not guarantee compatibility with all system calls, kernel modules, or filesystem operations.	We use security profiles (`seccomp` , `AppArmor`) to limit the attack surface. This inherently restricts some OS features. Our environment is tailored for application code, not system administration.

Architecture Decision Record: Foundational Isolation Technology

Decision: Use Linux Container Namespaces and cgroups as the Primary Isolation Mechanism (V1)

- **Context**: We require strong isolation between untrusted function codes in a multi-tenant Linux environment. The choice of isolation technology fundamentally impacts security, performance (cold start), and implementation complexity.
- **Options Considered**:
  1. **Linux Containers (namespaces, cgroups, seccomp)**: The industry-standard approach used by Docker, providing process, filesystem, and network isolation at the kernel level.
  2. **MicroVMs (Firecracker)**: Lightweight virtual machines that offer stronger hardware-enforced isolation, pioneered by AWS Lambda.
  3. **gVisor (User-space Kernel)**: A sandbox that intercepts application syscalls and implements them in a user-space kernel, providing a different security boundary.
- **Decision**: For V1, we will implement isolation using **Linux containers** (via a low-level library like `containerd` or direct `runc` calls), augmented with `seccomp` and `AppArmor` profiles.
- **Rationale**:
  - **Maturity & Ecosystem**: Containers have a vast ecosystem of tools, images, and knowledge. Debugging and operational tooling are readily available.
  - **Cold Start Performance**: Container startup, especially with optimized base images and shared layers, can be extremely fast (tens of milliseconds), which is critical for our cold start goals.
  - **Implementation Simplicity**: Managing containers is a well-understood problem. We can leverage existing battle-tested libraries, reducing our initial implementation risk.
  - **Sufficient Isolation for V1**: With careful hardening (dropping capabilities, applying strict `seccomp` filters, using read-only root filesystems), containers provide a robust level of isolation suitable for many multi-tenant scenarios.
- **Consequences**:
  - **Positive**: Faster path to a working V1, easier debugging, better community support.
  - **Negative**: The isolation boundary is the Linux kernel, which has a large attack surface compared to a MicroVM. A kernel vulnerability could potentially be exploited to break out of the container.
  - **Migration Path**: The design will abstract the isolation layer behind a `ContainerManager` interface, making a future migration to Firecracker or gVisor possible without changing the core runtime logic.

The following table compares the isolation options in more detail, highlighting why containers were chosen for our initial implementation:

Option	Pros	Cons	Applicability to Our Goals
Linux Containers	- Extremely fast startup (ms). - Mature, stable tooling (`runc`, `containerd`). - Fine-grained resource control via `cgroups`. - Rich security hardening via `seccomp`, `AppArmor`, capabilities.	- Isolation depends on a shared kernel. Kernel vulnerabilities can lead to container escape. - Requires host kernel configuration (namespaces, `cgroups`).	**CHOSEN**. Best aligns with our **cold start latency (G2)** and **implementation simplicity** goals for V1. Provides sufficient isolation for many use cases.
MicroVMs (Firecracker)	- Strong, hardware-enforced isolation (each function gets its own kernel). - Minimal attack surface (specially built kernel). - Excellent security story for hostile multi-tenancy.	- Higher cold start latency (~100-400ms) due to VM boot, though snapshotting helps. - More complex to implement and manage (requires KVM, device emulation). - Higher memory overhead per instance.	A strong candidate for a future V2 focused on maximum security, especially for public cloud offerings. The cold start trade-off is notable.
gVisor	- No need for hardware virtualization. - Syscall filtering provides a strong security boundary different from containers. - Good compatibility with many applications.	- Higher performance overhead for syscall-heavy workloads. - Some compatibility gaps with certain syscalls or applications. - Cold start similar to containers.	An interesting middle ground. However, the performance characteristics and maturity were deemed less certain than pure containers for our V1.

Common Pitfalls in Scope Definition

⚠️ Pitfall: Under-Scoping (The "Just One More Feature" Trap)

- **Description**: Continuously adding "small" features from the Non-Goals list because they seem easy or are requested by early users.
- **Why it's Wrong**: It leads to a bloated, complex V1 that never ships. It diverts effort from perfecting the core goals (like cold start optimization) and creates a confusing product identity.

- **How to Avoid**: Be militant about the Non-Goals list. New features must be explicitly justified against the core goals. Create a "Future Extensions" backlog for ideas that don't fit V1.

⚠️ **Pitfall: Over-Scoping (The "All-or-Nothing" Isolation Fallacy)**

- **Description**: Believing that because containers don't offer perfect, theoretical isolation, they are unacceptable, and therefore we must start with the most secure option (MicroVMs), regardless of complexity.
- **Why it's Wrong**: It ignores pragmatic trade-offs. The risk profile of many multi-tenant environments (e.g., internal company platforms, trusted communities) is well-managed by hardened containers. Starting with MicroVMs could delay the project by months and make cold start optimization much harder.
- **How to Avoid**: Conduct a realistic threat model for your target users. Choose the simplest isolation technology that mitigates the actual identified threats. Plan for iterative improvement.

⚠️ **Pitfall: Vague Success Criteria**

- **Description**: Defining goals with ambiguous terms like "fast," "scalable," or "reliable" without quantifiable metrics.
- **Why it's Wrong**: It makes it impossible to objectively determine if the project is successful. It leads to debates and moving goalposts.
- **How to Avoid**: Every goal must have a **measurable** success criterion, as shown in the Goals table. "Sub-second cold start (P95)" is measurable. "Fast cold start" is not.

Implementation Guidance

While this section is primarily conceptual, establishing clear goals and non-goals has direct implications for our code structure and testing strategy from the very beginning.

A. Technology Recommendations Table

Component	Simple Option (V1)	Advanced Option (Future)	Rationale for V1 Choice
Isolation Layer	`containerd` Go client (or direct `runc`)	Firecracker Go SDK (`firecracker-go-sdk`)	`containerd` is the industry-standard container runtime, providing a stable API and handling low-level details. It's the path of least resistance.
Resource Limits	Linux `cgroups` v2 (via `containerd` or `github.com/containerd/cgroups`)	Custom cgroup manager + quota enforcement	Letting `containerd` manage cgroups simplifies setup and ensures compatibility.
Security Hardening	Default `seccomp` profile (Docker's) + dropped capabilities + read-only rootfs	Custom AppArmor profiles, SELinux policies	Starting with well-known, conservative defaults is safe. Custom policies can be added later based on need.
Metrics & Observability	Prometheus client library (`github.com/prometheus/client_golang`) + structured logging (`logrus` / `zap`)	OpenTelemetry SDK for distributed tracing	Prometheus is the de facto standard for metrics. Structured logs are essential for debugging.

B. Recommended File/Module Structure

The goals and non-goals inform a clean separation of concerns from the project's inception. Here is the recommended high-level directory structure:

```
serverless-runtime/
├── cmd/
│   ├── controller/          # Main orchestrator (scaling, instance management)
│   │   └── main.go
│   ├── gateway/             # HTTP gateway (request routing)
│   │   └── main.go
│   └── cli/                 # Command-line tool for deploying functions
│       └── main.go
├── internal/                # Private application code
│   ├── api/                 # Internal APIs and data types
│   │   ├── types.go         # Core types: FunctionDefinition, InvocationRequest, etc.
│   │   └── storage.go       # Storage backend interface
│   ├── builder/             # **Milestone 1**: Function packaging & dependency bundling
│   │   ├── go_bundler.go
│   │   ├── python_bundler.go
│   │   └── node_bundler.go
│   ├── container/           # **Milestone 2 & 3**: Isolation & cold start optimization
│   │   ├── manager.go       # ContainerManager interface
│   │   ├── cgroup_utils.go  # Helper for cgroups
│   │   ├── warm_pool.go     # WarmPoolManager
│   │   └── criu_helper.go   # Snapshot/restore logic (if implemented)
│   ├── gateway/             # **Milestone 4**: Request routing
│   │   ├── router.go        HTTP routing logic
│   │   ├── queue.go         # Request queue implementation
│   │   └── loadbalancer.go  # Instance selection logic
│   ├── scaler/              # **Milestone 5**: Auto-scaling
│   │   ├── metrics.go       # Collects RPS, concurrency
│   │   └── decision.go      # Scaling algorithm
│   └── store/               # Storage implementations
│       ├── s3_store.go      # S3 backend for function artifacts
│       └── local_store.go   # Local disk backend for development
├── pkg/                     # Public, reusable libraries (if any)
│   └── function_spec/       # Public definition of func.yaml format
├── go.mod
└── README.md
```

C. Core Logic Skeleton: Goal Validation Metric

To embody the principle of measurable goals, we can create a simple metric collection point from day one. This code doesn't implement the logic but sets up the scaffolding for tracking our key success metric: cold start latency.

```go
// internal/api/metrics.go

package api


import (

	"time"

	"github.com/prometheus/client_golang/prometheus"

	"github.com/prometheus/client_golang/prometheus/promauto"

)


// Metrics is a centralized struct holding all Prometheus metrics for the runtime.

// This ensures we track the data needed to validate our Goals (G2, G4, G7, G8).

type Metrics struct {

	// ColdStartLatency tracks the duration from when a request is received

	// for a function with no warm instance to when the function code begins execution.

	// This is our KEY metric for Goal G2.

	ColdStartLatency prometheus.Histogram


	// InvocationDuration tracks total function execution time.

	InvocationDuration prometheus.Histogram


	// ActiveInvocations is a gauge of currently running functions.

	ActiveInvocations prometheus.Gauge


	// InvocationErrors counts failures by type (timeout, OOM, crash).

	InvocationErrors *prometheus.CounterVec


	// FunctionInvocations counts total invocations per function.

	FunctionInvocations *prometheus.CounterVec

}


// NewMetrics creates and registers all metrics. Call this once at startup.

func NewMetrics() *Metrics {

	return &Metrics{

		ColdStartLatency: promauto.NewHistogram(prometheus.HistogramOpts{

			Name:    "function_cold_start_latency_seconds",

			Help:    "Time from cold invocation request to code execution start.",

			Buckets: prometheus.DefBuckets, // Default buckets are good for sub-second latencies

		}),

		InvocationDuration: promauto.NewHistogram(prometheus.HistogramOpts{

			Name:    "function_invocation_duration_seconds",

			Help:    "Total duration of function execution.",

			Buckets: prometheus.DefBuckets,

		}),

		ActiveInvocations: promauto.NewGauge(prometheus.GaugeOpts{
```

```
                        Name: "function_active_invocations",
                        Help: "Current number of concurrently executing function invocations.",
                }),
                InvocationErrors: promauto.NewCounterVec(prometheus.CounterOpts{
                        Name: "function_invocation_errors_total",
                        Help: "Total number of failed function invocations by error type.",
                }, []string{"function", "error_type"}), // error_type: "timeout", "oom", "crash", "system"
                FunctionInvocations: promauto.NewCounterVec(prometheus.CounterOpts{
                        Name: "function_invocations_total",
                        Help: "Total number of function invocations.",
                }, []string{"function"}),
        }
}

// RecordColdStart should be called when a cold start is completed.
// `startTime` should be the time the request was first routed (before acquiring an instance).
func (m *Metrics) RecordColdStart(startTime time.Time) {
        duration := time.Since(startTime).Seconds()
        m.ColdStartLatency.Observe(duration)
}

// RecordInvocation should be called when a function invocation finishes.
func (m *Metrics) RecordInvocation(functionName string, startTime time.Time, err error) {
        duration := time.Since(startTime).Seconds()
        m.InvocationDuration.Observe(duration)
        m.FunctionInvocations.WithLabelValues(functionName).Inc()
        m.ActiveInvocations.Dec() // Assuming we .Inc() when starting an invocation

        if err != nil {
                // TODO: Classify the error type (timeout, OOM, etc.)
                errorType := classifyError(err)
                m.InvocationErrors.WithLabelValues(functionName, errorType).Inc()
        }
}

// classifyError is a helper to categorize errors for metrics.
// This is a simplistic example; real implementation would be more robust.
func classifyError(err error) string {
        // TODO: Implement error type classification based on your runtime's error types.
        // Example: check for context.DeadlineExceeded, or OOM messages from the container.
        return "unknown"
}
```

D. Language-Specific Hints (Go)

- **Prometheus Integration**: Use the official `prometheus/client_golang` library. The `promauto` package is convenient for automatically registering metrics.
- **Time Measurement**: Use `time.Now()` or `time.Since()` for latency measurements. For high precision, consider `time.Now().UnixNano()`.
- **Concurrency with Gauges**: Be careful to call `metrics.ActiveInvocations.Inc()` and `.Dec()` in a thread-safe manner, ideally within a synchronized request handler.

E. Milestone Checkpoint: Goal Validation

After implementing the initial scaffolding (even before full routing), you should be able to validate that your metrics infrastructure is working.

1. **Command**: Start your controller or gateway with the metrics code integrated. The Prometheus client will expose a `/metrics` HTTP endpoint by default on the same port as your application (if you use `promhttp.Handler()`).
2. **Expected Output**: Run `curl http://localhost:<port>/metrics`. You should see the raw Prometheus metrics, including the histograms and counters defined above (their values will be zero initially).
3. **Sign of Success**: The metric names appear correctly in the output. If you see `function_cold_start_latency_seconds` listed, your instrumentation is correctly registered.
4. **Troubleshooting**: If metrics are missing, ensure you've called `NewMetrics()` and that the metrics variables are not garbage collected (they should be in a long-lived struct). Also verify the Prometheus handler is registered with your HTTP server: `http.Handle("/metrics", promhttp.Handler())`.

High-Level Architecture

> **Milestone(s):** This foundational section establishes the architectural blueprint that all subsequent milestones (1-5) will implement. It defines the major system components, their responsibilities, and how they interact to solve the core challenge of balancing isolation, speed, and efficient scaling.

Component Overview and Responsibilities

Mental Model: The Instant Kitchen Factory

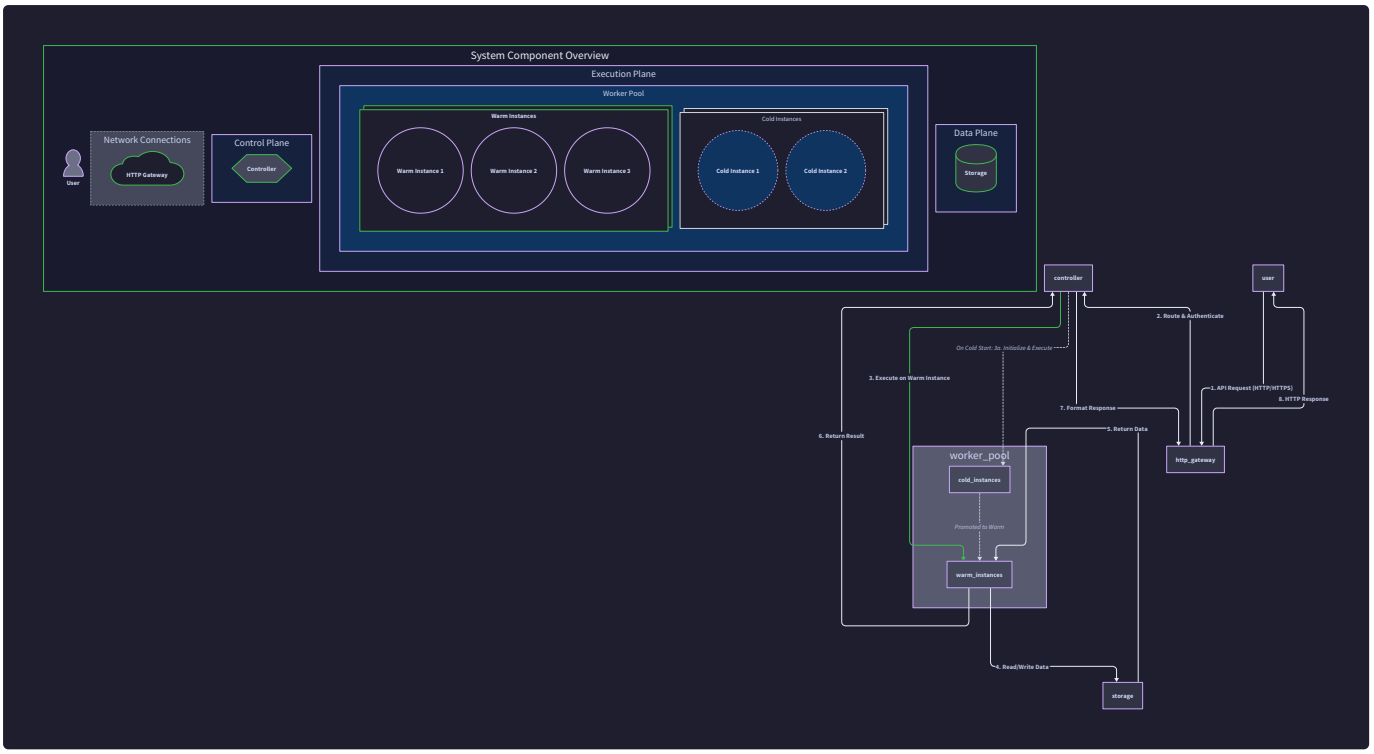
Imagine you're running a gourmet meal delivery service that needs to prepare thousands of unique dishes on demand. Each dish (function) requires specific ingredients (code + dependencies) and cooking equipment (runtime). You face three key challenges:

1. **Speed**: Customers want their meals in seconds, not hours
2. **Isolation**: You can't let the garlic from one dish contaminate the dessert
3. **Efficiency**: You can't keep every possible kitchen setup running 24/7 when demand fluctuates

Your solution is an "Instant Kitchen Factory" with these specialized departments:

- **The Order Desk (Gateway)**: Where customers place orders, get queued if all kitchens are busy, and receive their finished meals
- **The Control Room (Controller)**: The brain that tracks which kitchens are available, creates new ones when needed, and shuts down idle ones
- **The Kitchen Stations (Worker Pool)**: Individual, isolated kitchens where meals are actually prepared, each with its own equipment and ingredients
- **The Warehouse (Storage)**: Where recipe books (function code), pre-measured ingredient kits (dependencies), and cooking equipment (runtime images) are stored

This factory can instantly deploy a fully-equipped kitchen for any recipe, clean it up when done, and keep a few popular kitchens "warm" and ready to go. Now let's translate this mental model to our technical architecture.



The system comprises four core components that work together to provide serverless function execution. Each component is designed for specific responsibilities while maintaining clear boundaries and well-defined interfaces.

HTTP Gateway: The Public Entry Point

The Gateway serves as the single entry point for all function invocations, handling request routing, load balancing, and traffic management. Think of it as the restaurant host who greets customers, checks reservations (function names), and seats them at available tables (function instances).

Responsibility	Description
Request Reception	Accepts HTTP requests on a public endpoint (e.g., `POST /invoke/{functionName}`) with function input as request body
Request Validation	Validates incoming requests against function specifications (timeout limits, payload size) before forwarding
Function Resolution	Maps URL paths or headers to specific function names and versions (e.g., `/api/users/create` → `createUser:v1.2`)
Load Balancing	Distributes requests across available function instances using configurable strategies (round-robin, least connections)
Request Queuing	Maintains per-function queues when all instances are busy, applying backpressure and fairness policies
Synchronous/Async Mode	Supports immediate response (synchronous) and fire-and-forget (asynchronous) invocation patterns
Response Aggregation	Collects function output (or error) and returns appropriate HTTP status codes and headers to the caller
Request Timeout Enforcement	Terminates requests that exceed configured timeouts, returning appropriate error responses

The Gateway maintains minimal state—primarily request queues and circuit breakers—and delegates instance management to the Controller. It communicates with the Controller via gRPC or HTTP for instance assignment.

Controller: The Orchestration Brain

The Controller is the central coordination point that manages function lifecycle, scaling decisions, and system state. Imagine it as the air traffic control tower that monitors all aircraft (function instances), decides when to launch new ones, and directs them to landing spots (request assignments).

Responsibility	Description
Function Deployment	Processes function uploads, coordinates packaging with Storage, and registers new function versions
Instance Lifecycle Management	Creates, monitors, and destroys function instances (`FunctionInstance` objects) throughout their lifecycle
Warm Pool Management	Maintains pre-initialized instances ready for immediate execution, applying eviction and refresh policies
Auto-Scaling Decisions	Analyzes metrics (requests/second, concurrent executions) to scale instances up or down per function
Health Monitoring	Periodically checks instance health via heartbeat mechanisms and replaces unhealthy instances
System Metrics Collection	Aggregates performance data (cold starts, execution times, errors) for observability and scaling decisions
Configuration Management	Stores and applies function-specific configurations (timeouts, memory limits, environment variables)
Instance Assignment	Provides available instances to the Gateway upon request, considering load distribution and locality

The Controller maintains the authoritative view of system state, including all function definitions, active instances, and scaling policies. It uses the `ContainerManager` interface to manage execution environments and interacts with Storage for artifact retrieval.

Worker Pool: The Execution Fabric

The Worker Pool represents the actual compute resources where functions execute—a collection of isolated environments (containers or microVMs) running on physical or virtual hosts. Think of each worker as a self-contained kitchen with its own stove, sink, and pantry, completely isolated from other kitchens.

Responsibility	Description
Isolation Enforcement	Provides secure multi-tenant isolation using containers, namespaces, cgroups, and security profiles
Resource Management	Enforces CPU, memory, disk, and network limits for each function instance
Function Execution	Loads function code into the isolated environment and executes the handler with provided input
Runtime Lifecycle	Manages runtime-specific initialization (Python interpreter, JVM, Go binary) before function execution
Local Caching	Caches frequently used function artifacts and runtime layers to reduce startup latency
Metrics Reporting	Reports per-instance resource usage (CPU%, memory MB) and execution metrics to Controller
Cleanup	Ensures complete cleanup of execution environments after function completion or timeout

The Worker Pool isn't a centralized service but a distributed capability—each host runs a worker agent that communicates with the Controller. Multiple workers can run on a single host for density, but each function instance runs in its own isolated environment.

Storage: The Durable Backbone

Storage provides durable, scalable persistence for function artifacts, system state, and operational data. Imagine it as the warehouse + filing cabinet that stores everything from recipe books (code) to order records (invocation logs).

Responsibility	Description
Function Artifact Storage	Stores packaged function code with dependencies, supporting versioning and immutability
Runtime Image Repository	Stores base runtime images (Python, Node.js, Go) and common dependency layers
System State Persistence	Optionally persists function definitions, scaling policies, and instance assignments (for fault tolerance)
Log Aggregation	Collects and stores function execution logs for debugging and audit purposes
Metrics Archival	Stores historical metrics for trend analysis and capacity planning
Snapshot Storage	Stores container snapshots for fast restoration (if using CRIU checkpoint/restore)

Storage is designed as a pluggable backend, supporting cloud object storage (AWS S3, Google Cloud Storage), container registries (Docker Hub, AWS ECR), and local filesystems for development.

Component Interaction Patterns:

The components follow a clear hierarchy and communication pattern:

1. **North-South Traffic (User Requests)**:

```
User → HTTP Gateway → Controller → Worker Pool → Function Execution
```

2. **East-West Traffic (Control Plane)**:

```
Controller ↔ Storage (for artifacts)
Controller ↔ Worker Pool (for lifecycle management)
Gateway ↔ Controller (for instance assignment)
```

3. **Data Flow**:
   - Control signals flow from Controller to Workers
   - Metrics flow from Workers to Controller
   - Artifacts flow from Storage to Workers
   - Requests flow from Gateway to Workers (via Controller assignment)

> **Design Insight:** We deliberately separate the data plane (Gateway → Worker) from the control plane (Controller ↔ Worker). This allows the Gateway to communicate directly with Workers for low-latency request/response while the Controller manages orchestration without being in the hot path.

Recommended File and Module Structure

A well-organized codebase is critical for maintaining a complex system like this. Following Go conventions and separation of concerns, we recommend this project structure:

Mental Model: The Office Building

Think of the project structure as a well-organized office building:

- `/cmd`: The reception desks and main entrances (executable programs)
- `/internal`: The specialized departments with restricted access (private implementation)
- `/pkg`: The public-facing services anyone can visit (reusable libraries)
- `/api`: The building blueprints and interface specifications (contracts)
- `/deployments`: The building maintenance plans and utility hookups (infrastructure)

Here's the complete structure:

```
serverless-runtime/
├── cmd/                          # Application entry points
│   ├── gateway/                  # HTTP Gateway service
│   │   ├── main.go               # Gateway entry point with CLI flags
│   │   └── config/               # Gateway-specific configuration
│   ├── controller/               # Controller service
│   │   ├── main.go               # Controller entry point
│   │   └── config/               # Controller configuration
│   └── worker/                   # Worker agent (optional separate binary)
│       └── main.go               # Worker entry point
├── internal/                     # Private application code
│   ├── gateway/                  # Gateway implementation
│   │   ├── gateway.go            # Main Gateway struct and initialization
│   │   ├── router.go             # HTTP routing and middleware
│   │   ├── handler/              # HTTP request handlers
│   │   │   ├── invoke.go         # Function invocation handler
│   │   │   ├── async.go          # Async invocation handler
│   │   │   └── health.go         # Health check endpoint
│   │   ├── queue/                # Request queuing subsystem
│   │   │   ├── manager.go        # Queue manager interface
│   │   │   ├── memory_queue.go   # In-memory queue implementation
│   │   │   └── priority_queue.go # Priority-based queuing
│   │   ├── balancer/             # Load balancing logic
│   │   │   ├── balancer.go       # LoadBalancer interface
│   │   │   ├── round_robin.go    # Round-robin implementation
│   │   │   └── least_conn.go     # Least-connections implementation
│   │   └── client/               # Client to talk to Controller
│   │       └── controller_client.go # gRPC/HTTP client for Controller
│   ├── controller/               # Controller implementation
│   │   ├── controller.go         # Main Controller struct and coordination
│   │   ├── manager/              # Core management subsystems
│   │   │   ├── function/         # Function lifecycle management
│   │   │   │   ├── manager.go    # FunctionManager interface
│   │   │   │   ├── registry.go   # In-memory function registry
│   │   │   │   └── deployer.go   # Function deployment logic
│   │   │   ├── instance/         # Instance lifecycle management
│   │   │   │   ├── manager.go    # InstanceManager interface
│   │   │   │   ├── lifecycle.go  # State transitions and lifecycle
│   │   │   │   └── health.go     # Health checking logic
│   │   │   └── warmpool/         # Warm pool management
│   │   │       ├── manager.go    # WarmPoolManager interface
│   │   │       ├── pool.go       # Pool implementation
│   │   │       └── eviction.go   # Eviction policies (LRU, LFU)
│   │   ├── scaler/               # Auto-scaling logic
│   │   │   ├── scaler.go         # Scaler interface
│   │   │   ├── reactive.go       # Reactive scaling implementation
│   │   │   ├── metrics/          # Metrics collection for scaling
│   │   │   │   ├── collector.go  # Metrics collector interface
│   │   │   │   └── prometheus.go # Prometheus implementation
│   │   │   └── decision/         # Scaling decision logic
│   │   │       ├── engine.go     # Decision engine
│   │   │       └── policies.go   # Scaling policies
│   │   ├── api/                  # Controller API (gRPC/HTTP)
│   │   │   ├── server.go         # API server implementation
│   │   │   ├── service.go        # gRPC service definition
│   │   │   └── handlers/         # HTTP handlers (if dual protocol)
│   │   └── storage/              # Controller's storage client
│   │       └── client.go         # Client to Storage service
│   ├── worker/                   # Worker implementation
│   │   ├── runtime/              # Runtime environment management
│   │   │   ├── manager.go        # RuntimeManager interface
│   │   │   ├── container/        # Container-based runtime
│   │   │   │   ├── manager.go    # Container runtime manager
│   │   │   │   └── executor.go   # Container execution logic
│   │   │   └── microvm/          # MicroVM-based runtime (optional)
│   │   │       ├── manager.go    # Firecracker manager
│   │   │       └── executor.go   # MicroVM execution logic
│   │   ├── invoker/              # Function invocation
│   │   │   ├── invoker.go        # Invoker interface
│   │   │   ├── http_invoker.go   # HTTP-triggered invocation
│   │   │   └── event_invoker.go  # Event-triggered invocation
│   │   ├── isolation/            # Isolation setup
│   │   │   ├── cgroups.go        # cgroups setup and management
│   │   │   ├── namespaces.go     # Linux namespace setup
│   │   │   └── security.go       # seccomp/AppArmor profiles
│   │   └── snapshot/             # Snapshot/restore functionality
│   │       ├── criu.go           # CRIU integration
│   │       └── snapshotter.go    # Snapshot manager
│   ├── packaging/                # Function packaging subsystem
│   │   ├── bundler/              # Language-specific bundlers
```

```
│   │   │   ├── bundler.go        # Bundler interface
│   │   │   ├── go_bundler.go     # Go bundler implementation
│   │   │   ├── python_bundler.go # Python bundler
│   │   │   └── node_bundler.go   # Node.js bundler
│   │   ├── registry/             # Function registry and storage
│   │   │   ├── storage.go        # Storage interface for artifacts
│   │   │   ├── s3_storage.go     # S3-backed storage
│   │   │   └── local_storage.go  # Local filesystem storage
│   │   └── builder/              # Build system integration
│   │       └── builder.go        # Build orchestration
│   ├── models/                   # Shared data structures
│   │   ├── function.go           # FunctionDefinition and related types
│   │   ├── instance.go           # FunctionInstance and states
│   │   ├── invocation.go         # InvocationRequest and response types
│   │   ├── container.go          # Container and ContainerStatus
│   │   └── metrics.go            # Metrics data structures
│   └── util/                     # Shared utilities
│       ├── logging/              # Structured logging
│       ├── config/               # Configuration parsing
│       ├── retry/                # Retry utilities
│       └── telemetry/            # Tracing and telemetry
├── pkg/                          # Public reusable packages
│   ├── container/                # Container management abstraction
│   │   ├── manager.go            # ContainerManager interface
│   │   ├── types.go              # Container, CreateContainerRequest, etc.
│   │   └── docker/               # Docker implementation
│   │       ├── docker_manager.go # DockerManager implementation
│   │       └── client.go         # Docker client wrapper
│   └── metrics/                  # Metrics collection utilities
│       ├── metrics.go            # Metrics struct and factory
│       └── prometheus.go         # Prometheus implementation
├── api/                          # API definitions and contracts
│   ├── v1/                       # Version 1 API
│   │   ├── gateway/              # Gateway API definitions
│   │   │   ├── gateway.proto     # gRPC service definition
│   │   │   └── openapi.yaml     # OpenAPI/Swagger spec
│   │   ├── controller/           # Controller API definitions
│   │   │   └── controller.proto # Controller gRPC service
│   │   └── models/               # Shared message types
│   │       └── models.proto      # Common protobuf messages
│   └── buf.yaml                  # buf.build configuration
├── deployments/                  # Deployment configurations
│   ├── docker/                   # Docker-related files
│   │   ├── Dockerfile.gateway    # Gateway Dockerfile
│   │   ├── Dockerfile.controller # Controller Dockerfile
│   │   └── docker-compose.yml    # Local development setup
│   └── kubernetes/               # Kubernetes manifests
│       ├── namespace.yaml        # K8s namespace
│       ├── gateway/              # Gateway deployment
│       ├── controller/           # Controller deployment
│       └── worker/               # Worker DaemonSet
├── scripts/                      # Build and development scripts
│   ├── build.sh                  # Build script
│   ├── test.sh                   # Test script
│   └── deploy/                   # Deployment scripts
├── configs/                      # Configuration files
│   ├── gateway.yaml.example      # Gateway config example
│   ├── controller.yaml.example   # Controller config example
│   └── worker.yaml.example       # Worker config example
├── tests/                        # Integration and e2e tests
│   ├── integration/              # Integration tests
│   └── e2e/                      # End-to-end tests
├── docs/                         # Documentation
│   ├── architecture/             # Architecture diagrams and docs
│   └── api/                      # API documentation
├── go.mod                        # Go module definition
├── go.sum                        # Go dependencies checksum
├── Makefile                      # Common tasks automation
└── README.md                     # Project overview
```

Key Organizational Principles:

1. **Clear Separation of Concerns**: Each component lives in its own directory with well-defined interfaces between them.
2. **Dependency Direction**:
   - Higher-level components (`gateway` , `controller`) depend on abstractions in `pkg/`
   - Lower-level implementations (`docker` , `s3_storage`) implement those abstractions
   - Dependencies flow inward: `cmd/` → `internal/` → `pkg/`

3. **Test Organization**: Unit tests live alongside the code they test (e.g., `gateway_test.go` next to `gateway.go`), while integration and end-to-end tests are separate.
4. **Configuration Management**: Each service has its own configuration structure, with examples provided and environment-specific overrides.
5. **API-First Design**: Protobuf/OpenAPI definitions are the source of truth for external APIs, enabling code generation for multiple languages.

> **Design Insight:** This structure supports incremental development across milestones. You can implement the `packaging/` directory for Milestone 1, `worker/runtime/` for Milestone 2, `worker/snapshot/` for Milestone 3, `gateway/queue/` for Milestone 4, and `controller/scaler/` for Milestone 5—each in isolation while maintaining clear interfaces.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option (for learning/getting started)	Advanced Option (production-ready)
Gateway Transport	HTTP/1.1 with Go's `net/http`	HTTP/2 with gRPC-gateway for dual REST/gRPC
Controller-Worker Communication	HTTP REST with JSON	gRPC streaming for real-time control
Storage Backend	Local filesystem with directory structure	S3-compatible object storage with versioning
Container Runtime	Docker Engine via Docker SDK	containerd with CRI plugin for lower overhead
Metrics	Prometheus client library with in-memory aggregation	OpenTelemetry with Jaeger/Datadog export
Service Discovery	Static configuration in YAML	Consul or etcd for dynamic registration
Queue Implementation	In-memory Go channels with buffering	Redis Streams or Apache Kafka for persistence

B. Core Infrastructure Starter Code

Since the high-level architecture establishes the foundation, here's complete starter code for the shared models and interfaces that all components will use:

File: `internal/models/function.go`

```go
package models

import (
    "time"
)

// FunctionDefinition represents a deployable function with its configuration
type FunctionDefinition struct {
    // Unique identifier for the function (e.g., "user-auth")
    Name string `json:"name" yaml:"name"`

    // Runtime environment (e.g., "go1.19", "python3.9", "nodejs18")
    Runtime string `json:"runtime" yaml:"runtime"`

    // Handler entry point (e.g., "main.Handler", "index.handler")
    Handler string `json:"handler" yaml:"handler"`

    // Path to the function code in storage
    CodePath string `json:"codePath" yaml:"codePath"`

    // Maximum execution time in seconds
    Timeout int `json:"timeout" yaml:"timeout"`

    // Memory limit in megabytes
    MemoryMB int `json:"memoryMB" yaml:"memoryMB"`

    // CPU quota as percentage of a single core (100 = full core)
    CPUQuota int64 `json:"cpuQuota" yaml:"cpuQuota"`

    // Environment variables to inject
    EnvVars map[string]string `json:"envVars" yaml:"envVars"`

    // Language-specific dependencies
    Dependencies []string `json:"dependencies" yaml:"dependencies"`

    // Metadata
    Version    string    `json:"version" yaml:"version"`
    CreatedAt  time.Time `json:"createdAt" yaml:"createdAt"`
    UpdatedAt  time.Time `json:"updatedAt" yaml:"updatedAt"`
}

// FunctionInstance represents a running instance of a function
```

```go
type FunctionInstance struct {
    // Unique instance identifier
    ID string `json:"id" yaml:"id"`

    // Function this instance belongs to
    FunctionName    string `json:"functionName" yaml:"functionName"`
    FunctionVersion string `json:"functionVersion" yaml:"functionVersion"`

    // Current state (see constants below)
    State FunctionInstanceState `json:"state" yaml:"state"`

    // Underlying container/microVM identifier
    ContainerID string `json:"containerId" yaml:"containerId"`

    // Host where this instance is running
    Host string `json:"host" yaml:"host"`

    // Port for HTTP communication (if applicable)
    Port int `json:"port" yaml:"port"`

    // Resource usage tracking
    MemoryUsageMB int `json:"memoryUsageMB" yaml:"memoryUsageMB"`
    CPUUsage      float64 `json:"cpuUsage" yaml:"cpuUsage"`

    // Lifecycle timestamps
    CreatedAt   time.Time `json:"createdAt" yaml:"createdAt"`
    LastUsedAt  time.Time `json:"lastUsedAt" yaml:"lastUsedAt"`
    TerminatedAt *time.Time `json:"terminatedAt,omitempty" yaml:"terminatedAt,omitempty"`

    // Statistics
    InvocationCount int `json:"invocationCount" yaml:"invocationCount"`
    TotalDurationMS int64 `json:"totalDurationMS" yaml:"totalDurationMS"`
}

// FunctionInstanceState represents the lifecycle state of a function instance
type FunctionInstanceState string

const (
    // InstanceStateProvisioning - Container is being created/started
    InstanceStateProvisioning FunctionInstanceState = "PROVISIONING"
```

```
    // InstanceStateWarm - Container is running and ready for requests
    InstanceStateWarm FunctionInstanceState = "WARM"

    // InstanceStateActive - Container is currently executing a request
    InstanceStateActive FunctionInstanceState = "ACTIVE"

    // InstanceStateDraining - Container is finishing current requests, no new requests
    InstanceStateDraining FunctionInstanceState = "DRAINING"

    // InstanceStateTerminated - Container has been stopped and cleaned up
    InstanceStateTerminated FunctionInstanceState = "TERMINATED"

    // InstanceStateError - Container is in an error state and needs recovery
    InstanceStateError FunctionInstanceState = "ERROR"
)

// Validate checks if the FunctionDefinition has required fields
func (fd *FunctionDefinition) Validate() error {
    // TODO 1: Check that Name is non-empty and follows naming conventions
    // TODO 2: Validate Runtime is supported (go, python, nodejs, etc.)
    // TODO 3: Validate Handler format based on runtime
    // TODO 4: Ensure Timeout is between 1 and maximum allowed (e.g., 900 seconds)
    // TODO 5: Ensure MemoryMB is within allowed range (e.g., 128MB to 3008MB)
    // TODO 6: Validate CPUQuota is positive and within limits
    return nil
}

// IsActive returns true if the instance can accept new requests
func (fi *FunctionInstance) IsActive() bool {
    return fi.State == InstanceStateWarm || fi.State == InstanceStateActive
}

// Age returns how long the instance has been running
func (fi *FunctionInstance) Age() time.Duration {
    return time.Since(fi.CreatedAt)
}

// IdleDuration returns how long the instance has been idle
func (fi *FunctionInstance) IdleDuration() time.Duration {
    return time.Since(fi.LastUsedAt)
}
```

File: `internal/models/invocation.go`

```go
package models

import (
    "time"
)

// InvocationRequest represents a request to execute a function
type InvocationRequest struct {
    // Unique identifier for this invocation
    ID string `json:"id" yaml:"id"`

    // Function to invoke
    FunctionName    string `json:"functionName" yaml:"functionName"`
    FunctionVersion string `json:"functionVersion" yaml:"functionVersion"`

    // Input payload (JSON, binary, etc.)
    Payload []byte `json:"payload" yaml:"payload"`

    // Synchronous (wait for response) vs asynchronous (fire and forget)
    Synchronous bool `json:"synchronous" yaml:"synchronous"`

    // When this request must be completed by
    Deadline time.Time `json:"deadline" yaml:"deadline"`

    // Metadata
    RequestID string    `json:"requestId" yaml:"requestId"` // External request ID
    Source    string    `json:"source" yaml:"source"`       // Source of invocation (HTTP, event, etc.)
    CreatedAt time.Time `json:"createdAt" yaml:"createdAt"`

    // Routing hints (optional)
    PreferredInstanceID string `json:"preferredInstanceId,omitempty" yaml:"preferredInstanceId,omitempty"`

    // Callback URL for async invocations
    CallbackURL string `json:"callbackUrl,omitempty" yaml:"callbackUrl,omitempty"`
}

// InvocationResponse represents the result of a function execution
type InvocationResponse struct {
    // Corresponding request ID
    RequestID string `json:"requestId" yaml:"requestId"`

    // Function output
```

```
    Payload []byte `json:"payload" yaml:"payload"`

    // Execution metadata
    DurationMS   int64     `json:"durationMS" yaml:"durationMS"`
    MemoryUsedMB int       `json:"memoryUsedMB" yaml:"memoryUsedMB"`
    BilledDurationMS int64 `json:"billedDurationMS" yaml:"billedDurationMS"`

    // Error information (if any)
    Error     string `json:"error,omitempty" yaml:"error,omitempty"`
    ErrorType string `json:"errorType,omitempty" yaml:"errorType,omitempty"` // Runtime, Timeout, etc.

    // Instance that executed this
    InstanceID string `json:"instanceId" yaml:"instanceId"`

    // Timestamps
    StartedAt  time.Time `json:"startedAt" yaml:"startedAt"`
    FinishedAt time.Time `json:"finishedAt" yaml:"finishedAt"`
}

// InvocationStatus represents the status of an asynchronous invocation
type InvocationStatus struct {
    RequestID   string               `json:"requestId" yaml:"requestId"`
    Status      InvocationStatusType `json:"status" yaml:"status"`
    Result      *InvocationResponse  `json:"result,omitempty" yaml:"result,omitempty"`
    LastUpdated time.Time            `json:"lastUpdated" yaml:"lastUpdated"`
}

type InvocationStatusType string

const (
    InvocationStatusPending   InvocationStatusType = "PENDING"
    InvocationStatusRunning   InvocationStatusType = "RUNNING"
    InvocationStatusCompleted InvocationStatusType = "COMPLETED"
    InvocationStatusFailed    InvocationStatusType = "FAILED"
    InvocationStatusTimeout   InvocationStatusType = "TIMEOUT"
)

// IsExpired returns true if the request has passed its deadline
func (ir *InvocationRequest) IsExpired() bool {
    return time.Now().After(ir.Deadline)
}

// TimeRemaining returns how much time is left before the deadline
```

```
func (ir *InvocationRequest) TimeRemaining() time.Duration {

    return ir.Deadline.Sub(time.Now())

}
```

File: `pkg/container/manager.go`

```go
package container

import (
    "context"
    "io"
    "time"
)

// ContainerManager defines the interface for managing isolated execution environments
type ContainerManager interface {
    // CreateContainer creates a new container with the specified configuration
    CreateContainer(ctx context.Context, req CreateContainerRequest) (*Container, error)

    // StartContainer begins execution of the container
    StartContainer(ctx context.Context, containerID string) error

    // ExecuteInContainer runs a command inside a running container
    ExecuteInContainer(ctx context.Context, containerID string, cmd []string, input io.Reader, output io.Writer) (int, error)

    // StopContainer gracefully stops a running container
    StopContainer(ctx context.Context, containerID string, timeout time.Duration) error

    // RemoveContainer cleans up container resources
    RemoveContainer(ctx context.Context, containerID string) error

    // GetContainerStats returns resource usage statistics
    GetContainerStats(ctx context.Context, containerID string) (*ContainerStats, error)

    // ListContainers returns all managed containers
    ListContainers(ctx context.Context, filter ListFilter) ([]Container, error)

    // HealthCheck verifies the container manager is operational
    HealthCheck(ctx context.Context) error
}

// CreateContainerRequest contains all parameters needed to create a container
type CreateContainerRequest struct {
    // Container image name and tag
    Image string `json:"image" yaml:"image"`

    // Command to run (overrides image's CMD)
    Cmd []string `json:"cmd" yaml:"cmd"`
```

```
    // Environment variables
    Env []string `json:"env" yaml:"env"`

    // Labels for metadata
    Labels map[string]string `json:"labels" yaml:"labels"`

    // CPU quota (percentage of a single core, 100 = full core)
    CPUQuota int64 `json:"cpuQuota" yaml:"cpuQuota"`

    // Memory limit in bytes
    MemoryLimit int64 `json:"memoryLimit" yaml:"memoryLimit"`

    // Network mode (bridge, host, none)
    NetworkMode string `json:"networkMode" yaml:"networkMode"`

    // Volume bindings (host:container:mode)
    Binds []string `json:"binds" yaml:"binds"`

    // Working directory inside container
    WorkingDir string `json:"workingDir" yaml:"workingDir"`

    // User to run as (uid:gid)
    User string `json:"user" yaml:"user"`

    // Security options (seccomp, AppArmor)
    SecurityOpts []string `json:"securityOpts" yaml:"securityOpts"`

    // Exposed ports
    ExposedPorts map[string]struct{} `json:"exposedPorts" yaml:"exposedPorts"`

    // Port bindings
    PortBindings map[string][]PortBinding `json:"portBindings" yaml:"portBindings"`
}

// Container represents a container instance
type Container struct {
    // Unique identifier
    ID string `json:"id" yaml:"id"`

    // Container image
```

```
    Image string `json:"image" yaml:"image"`

    // Current status
    Status ContainerStatus `json:"status" yaml:"status"`

    // Creation timestamp
    CreatedAt time.Time `json:"createdAt" yaml:"createdAt"`

    // Labels for metadata
    Labels map[string]string `json:"labels" yaml:"labels"`

    // Additional metadata
    Names []string `json:"names" yaml:"names"`
    Ports []Port   `json:"ports" yaml:"ports"`
}

// ContainerStatus represents the lifecycle state of a container
type ContainerStatus string

const (
    StatusCreated    ContainerStatus = "created"
    StatusRunning    ContainerStatus = "running"
    StatusPaused     ContainerStatus = "paused"
    StatusRestarting ContainerStatus = "restarting"
    StatusExited     ContainerStatus = "exited"
    StatusDead       ContainerStatus = "dead"
)

// ContainerStats contains resource usage statistics
type ContainerStats struct {
    CPUPercent    float64   `json:"cpuPercent" yaml:"cpuPercent"`
    MemoryUsage   int64     `json:"memoryUsage" yaml:"memoryUsage"`
    MemoryLimit   int64     `json:"memoryLimit" yaml:"memoryLimit"`
    MemoryPercent float64   `json:"memoryPercent" yaml:"memoryPercent"`
    NetworkRx     int64     `json:"networkRx" yaml:"networkRx"`
    NetworkTx     int64     `json:"networkTx" yaml:"networkTx"`
    BlockRead     int64     `json:"blockRead" yaml:"blockRead"`
    BlockWrite    int64     `json:"blockWrite" yaml:"blockWrite"`
    PIDs          int       `json:"pids" yaml:"pids"`
    ReadAt        time.Time `json:"readAt" yaml:"readAt"`
}
```

```go
// ListFilter allows filtering containers by criteria
type ListFilter struct {
    Labels map[string]string `json:"labels" yaml:"labels"`
    Status []ContainerStatus `json:"status" yaml:"status"`
}

// Port represents a network port mapping
type Port struct {
    IP          string `json:"ip" yaml:"ip"`
    PrivatePort uint16 `json:"privatePort" yaml:"privatePort"`
    PublicPort  uint16 `json:"publicPort" yaml:"publicPort"`
    Type        string `json:"type" yaml:"type"`
}

// PortBinding represents a port binding configuration
type PortBinding struct {
    HostIP   string `json:"hostIp" yaml:"hostIp"`
    HostPort string `json:"hostPort" yaml:"hostPort"`
}
```

File: `pkg/metrics/metrics.go`

```go
package metrics

import (
    "time"

    "github.com/prometheus/client_golang/prometheus"
)

// Metrics encapsulates all Prometheus metrics for the runtime
type Metrics struct {
    // Cold start latency histogram (seconds)
    ColdStartLatency prometheus.Histogram

    // Function invocation duration histogram (seconds)
    InvocationDuration prometheus.Histogram

    // Currently active invocations gauge
    ActiveInvocations prometheus.Gauge

    // Invocation errors counter (by error type)
    InvocationErrors *prometheus.CounterVec

    // Function invocations counter (by function name)
    FunctionInvocations *prometheus.CounterVec

    // Container operations counter
    ContainerOperations *prometheus.CounterVec

    // Resource usage gauges
    MemoryUsageMB     *prometheus.GaugeVec
    CPUUsagePercent   *prometheus.GaugeVec
    ContainerCount    *prometheus.GaugeVec
}

// NewMetrics creates and registers all Prometheus metrics for the runtime
func NewMetrics() *Metrics {
    m := &Metrics{
        ColdStartLatency: prometheus.NewHistogram(
            prometheus.HistogramOpts{
                Name:    "function_cold_start_latency_seconds",
                Help:    "Time taken for cold start initialization",
                Buckets: prometheus.ExponentialBuckets(0.1, 2, 10), // 100ms to ~51.2s
```

```
        },
    ),

    InvocationDuration: prometheus.NewHistogram(
        prometheus.HistogramOpts{
            Name:    "function_invocation_duration_seconds",
            Help:    "Time taken for function execution",
            Buckets: prometheus.ExponentialBuckets(0.01, 2, 12), // 10ms to ~20.48s
        },
    ),

    ActiveInvocations: prometheus.NewGauge(
        prometheus.GaugeOpts{
            Name: "active_invocations_total",
            Help: "Number of currently executing function invocations",
        },
    ),

    InvocationErrors: prometheus.NewCounterVec(
        prometheus.CounterOpts{
            Name: "invocation_errors_total",
            Help: "Total number of invocation errors by type",
        },
        []string{"function", "error_type"},
    ),

    FunctionInvocations: prometheus.NewCounterVec(
        prometheus.CounterOpts{
            Name: "function_invocations_total",
            Help: "Total number of function invocations",
        },
        []string{"function", "runtime", "status"},
    ),

    ContainerOperations: prometheus.NewCounterVec(
        prometheus.CounterOpts{
            Name: "container_operations_total",
            Help: "Total number of container operations",
        },
        []string{"operation", "status"},
```

```
        ),

        MemoryUsageMB: prometheus.NewGaugeVec(
            prometheus.GaugeOpts{
                Name: "container_memory_usage_mb",
                Help: "Memory usage of containers in MB",
            },
            []string{"function", "instance_id"},
        ),

        CPUUsagePercent: prometheus.NewGaugeVec(
            prometheus.GaugeOpts{
                Name: "container_cpu_usage_percent",
                Help: "CPU usage of containers as percentage",
            },
            []string{"function", "instance_id"},
        ),

        ContainerCount: prometheus.NewGaugeVec(
            prometheus.GaugeOpts{
                Name: "container_count_total",
                Help: "Number of containers by state",
            },
            []string{"function", "state"},
        ),
    }

    // Register all metrics
    prometheus.MustRegister(
        m.ColdStartLatency,
        m.InvocationDuration,
        m.ActiveInvocations,
        m.InvocationErrors,
        m.FunctionInvocations,
        m.ContainerOperations,
        m.MemoryUsageMB,
        m.CPUUsagePercent,
        m.ContainerCount,
    )
```

```
    return m
}

// RecordColdStart records the duration of a cold start invocation
func (m *Metrics) RecordColdStart(startTime time.Time) {
    duration := time.Since(startTime).Seconds()
    m.ColdStartLatency.Observe(duration)
}

// RecordInvocation records the completion of a function invocation
func (m *Metrics) RecordInvocation(functionName string, startTime time.Time, err error) {
    duration := time.Since(startTime).Seconds()
    m.InvocationDuration.Observe(duration)

    // Update counters based on success/error
    status := "success"
    if err != nil {
        status = "error"
        m.InvocationErrors.WithLabelValues(functionName, "execution_error").Inc()
    }

    m.FunctionInvocations.WithLabelValues(functionName, "", status).Inc()
}
```

C. Entry Point Skeleton Code

File: `cmd/controller/main.go`

```go
package main

import (
    "context"
    "log"
    "net/http"
    "os"
    "os/signal"
    "syscall"
    "time"

    "github.com/yourorg/serverless-runtime/internal/controller"
    "github.com/yourorg/serverless-runtime/pkg/metrics"
    "github.com/prometheus/client_golang/prometheus/promhttp"
)

func main() {
    // TODO 1: Load configuration from file and environment variables
    // TODO 2: Initialize structured logger with appropriate level

    // Create metrics instance
    metrics := metrics.NewMetrics()

    // Initialize controller
    // TODO 3: Create ContainerManager implementation (DockerManager)
    // TODO 4: Create Storage client
    // TODO 5: Create FunctionManager, InstanceManager, WarmPoolManager
    // TODO 6: Create Scaler with appropriate strategy
    // TODO 7: Initialize Controller with all dependencies

    ctrl, err := controller.NewController(controller.Config{
        // TODO 8: Populate configuration
    })
    if err != nil {
        log.Fatalf("Failed to create controller: %v", err)
    }

    // Start controller
    ctx, cancel := context.WithCancel(context.Background())
    defer cancel()
```

```
    // Start metrics server
    go func() {
        http.Handle("/metrics", promhttp.Handler())
        http.HandleFunc("/health", func(w http.ResponseWriter, r *http.Request) {
            // TODO 9: Implement health check that verifies all subsystems
            w.WriteHeader(http.StatusOK)
        })

        log.Println("Starting metrics server on :9090")
        if err := http.ListenAndServe(":9090", nil); err != nil {
            log.Printf("Metrics server error: %v", err)
        }
    }()

    // Start gRPC/HTTP API server
    // TODO 10: Start API server for Gateway communication

    // Start scaling decision loop
    // TODO 11: Start periodic scaling evaluation in goroutine

    // Start warm pool maintenance
    // TODO 12: Start periodic warm pool cleanup and refresh

    log.Println("Controller started successfully")

    // Wait for shutdown signal
    sigChan := make(chan os.Signal, 1)
    signal.Notify(sigChan, syscall.SIGINT, syscall.SIGTERM)

    <-sigChan
    log.Println("Shutdown signal received")

    // Graceful shutdown
    shutdownCtx, shutdownCancel := context.WithTimeout(context.Background(), 30*time.Second)
    defer shutdownCancel()

    // TODO 13: Implement graceful shutdown - stop accepting new requests,
    //          drain existing requests, clean up resources
    if err := ctrl.Shutdown(shutdownCtx); err != nil {
        log.Printf("Error during shutdown: %v", err)
```

```
    }


    log.Println("Controller shutdown complete")
}
```

D. Language-Specific Hints for Go

1. **Concurrency**: Use `sync.WaitGroup` for coordinating goroutine completion and `errgroup` for error propagation in concurrent operations.
2. **Context Propagation**: Always pass `context.Context` through function calls to enable cancellation and timeouts across the entire call chain.
3. **Error Handling**: Use `fmt.Errorf` with `%w` for wrapping errors and implement custom error types for specific error categories (e.g., `ErrFunctionNotFound`, `ErrInstanceBusy`).
4. **Configuration**: Consider using `viper` for configuration management with support for multiple formats (YAML, JSON, env vars) and `validator` for struct validation.
5. **HTTP Client Pooling**: Reuse HTTP clients with appropriate timeouts:

```go
httpClient := &http.Client{
    Timeout: 30 * time.Second,
    Transport: &http.Transport{
        MaxIdleConns:        100,
        MaxIdleConnsPerHost: 10,
        IdleConnTimeout:     90 * time.Second,
    },
}
```

6. **Graceful Shutdown Pattern**: Implement graceful shutdown using contexts and wait for in-flight operations to complete before exiting.
7. **Structured Logging**: Use `slog` (Go 1.21+) or `logrus` / `zap` for structured logging with fields for correlation IDs and request tracing.
8. **Metrics Collection**: Use the Prometheus client library and expose metrics on a separate port (typically `:9090`) for scraping by monitoring systems.
9. **Interface-Based Design**: Define clear interfaces between components to enable testing and alternative implementations (e.g., different `ContainerManager` implementations).
10. **Dependency Injection**: Use constructor-based dependency injection to make components testable and configurable.

E. Milestone Checkpoint for Architecture Foundation

After setting up the basic structure:

1. **Verify Project Structure**:

```
$ tree -L 3 serverless-runtime/
# Should show the directory structure as outlined above
```

2. **Build the Project**:

```
$ cd serverless-runtime
$ go mod init github.com/yourorg/serverless-runtime
$ go mod tidy
$ go build ./cmd/controller
$ go build ./cmd/gateway
```

3. **Run Basic Tests**:

```
$ go test ./internal/models/... -v
# Should pass model validation tests
```

4. **Check Dependencies**:

```
$ go list -m all | grep -E "(prometheus|docker|grpc)"
# Should show required dependencies
```

5. **Expected Signs of Correct Setup**:
   - All Go files compile without errors
   - Models package can be imported by other packages
   - Prometheus metrics can be registered without panic
   - Basic `FunctionDefinition` validation logic works
6. **Common Early Issues**:
   - **Symptom**: "Cannot find package" errors
     - **Fix**: Ensure `go.mod` is in the root directory and run `go mod tidy`
   - **Symptom**: Circular import dependencies
     - **Fix**: Re-evaluate package structure - higher-level packages should not import lower-level implementation packages
   - **Symptom**: Interface implementation errors
     - **Fix**: Ensure all interface methods are implemented with exact signatures

This architectural foundation sets the stage for implementing each milestone systematically. The clear separation of concerns allows you to work on packaging (Milestone 1), execution (Milestone 2), optimization (Milestone 3), routing (Milestone 4), and scaling (Milestone 5) in parallel or sequence with well-defined interfaces between components.

Data Model

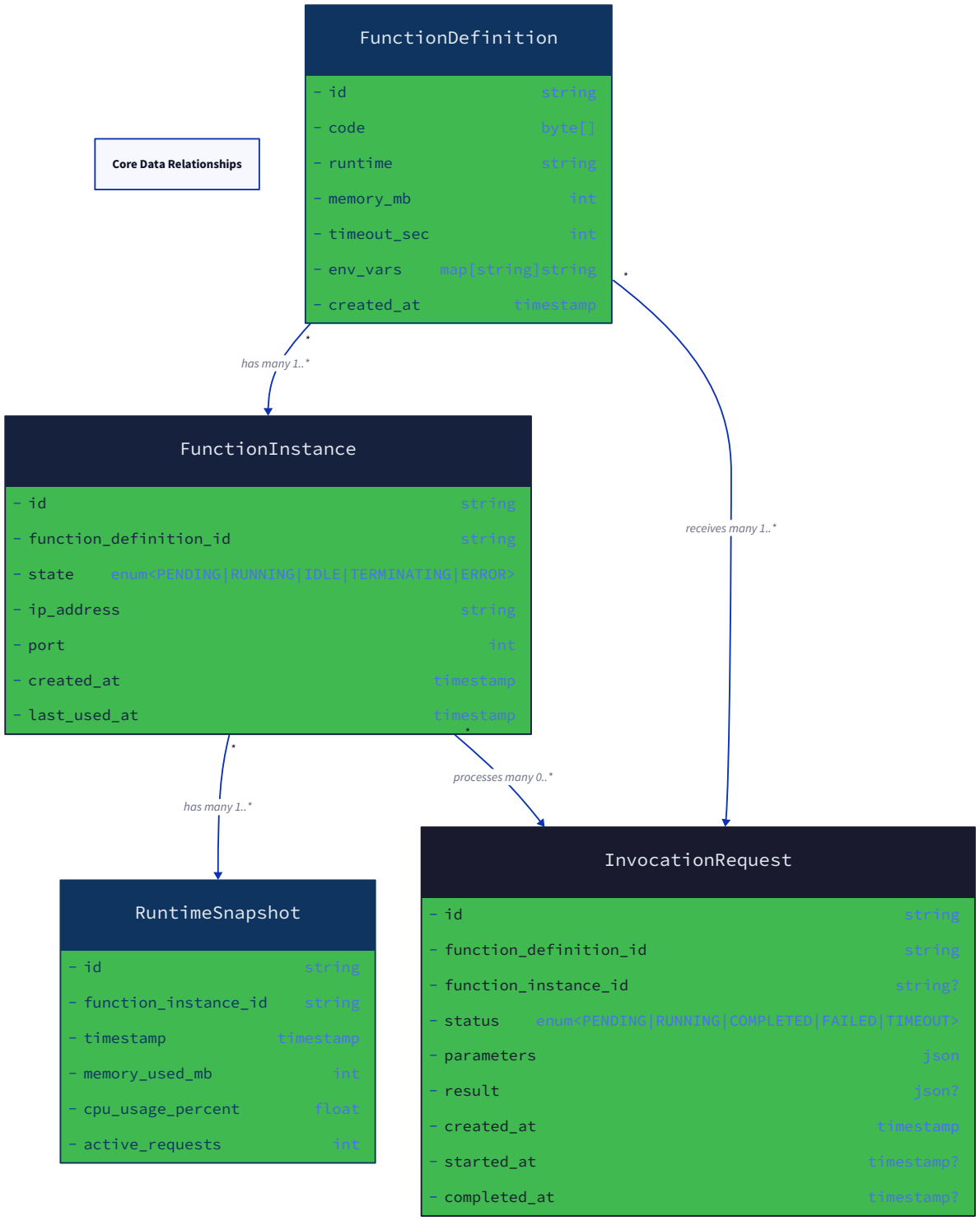
> **Milestone(s):** This section establishes the fundamental data structures that all subsequent milestones (1-5) will use to represent functions, their execution environments, and invocation requests. It provides the "vocabulary" for the entire system.

A serverless runtime's data model serves as the system's shared language—it defines what a function *is*, how an invocation *is represented*, and what state an execution environment *can be in*. Without a precise data model, components would pass ambiguous messages, leading to race conditions, resource leaks, and incorrect behavior. Think of this as the **blueprint and vocabulary** that every component (gateway, controller, worker pool) must agree upon to collaborate effectively.

This section defines the core entities and their relationships. First, we'll establish the key data structures through detailed tables, then formalize the lifecycle states that function instances transition through during their existence.

Core Types and Structures

The system revolves around three primary entities: **Function Definitions** (the code and configuration), **Function Instances** (running environments executing that code), and **Invocation Requests** (individual calls to execute the function). Their relationships can be visualized as:



A single `FunctionDefinition` can have many `FunctionInstance`s running simultaneously (for scaling). Each `FunctionInstance` processes one `InvocationRequest` at a time (though it may process multiple sequentially). An `InvocationRequest` targets a specific `FunctionDefinition` but may be routed to any available `FunctionInstance` for that function.

FunctionDefinition: The Blueprint

Think of a `FunctionDefinition` as a **recipe card** for your function. It contains all the instructions needed to create an execution environment: what code to run, which runtime to use, how much memory to allocate, and what environment variables to set. This is an immutable specification—once created, it represents a specific version of your function code and configuration.

Field	Type	Description
`Name`	`string`	Unique identifier for the function (e.g., "image-processor"). Used in API paths and logging.
`Runtime`	`string`	Execution environment language and version (e.g., "python3.9", "go1.19", "nodejs18"). Determines which base container image to use.
`Handler`	`string`	Entry point within the code package. Format varies by runtime: Python uses "module.function", Go uses compiled binary name, Node.js uses "module.export".
`CodePath`	`string`	Storage location URI for the packaged function artifact (e.g., "s3://bucket/functions/image-processor-v1.zip"). References the actual code bytes.
`Timeout`	`int`	Maximum execution duration in seconds before the function is forcibly terminated. Critical for preventing runaway executions.
`MemoryMB`	`int`	Memory allocation in megabytes. This determines both the container's memory limit and billing granularity in production systems.
`CPUQuota`	`int64`	CPU allocation as a percentage of a single core (100 = 1 core, 200 = 2 cores). Used to set cgroup CPU quotas for fair sharing.
`EnvVars`	`map[string]string`	Key-value pairs injected as environment variables into the execution environment. Used for configuration like API keys or feature flags.
`Dependencies`	`[]string`	List of dependency package names/versions (language-specific). For Python: `["requests==2.28.0", "pillow>=9.0"]`. For Go: empty (dependencies compiled in).

> **Key Insight:** The `FunctionDefinition` is **immutable** once stored. When you update a function, you create a new `FunctionDefinition` with a new version identifier, leaving previous versions intact for rollback and audit purposes.

FunctionInstance: The Running Process

A `FunctionInstance` represents an **actual, running execution environment**—a container or microVM that has been instantiated from a `FunctionDefinition`. Think of it as a **kitchen that has been set up according to a recipe card**. It has its own resources, lifecycle, and current state.

Field	Type	Description
`ID`	`string`	Unique identifier for this specific instance (UUID). Used for routing, logging, and management.
`FunctionName`	`string`	References the `FunctionDefinition.Name` this instance was created from.
`FunctionVersion`	`string`	References the specific version of the function (e.g., "v1.2.3"). Together with `FunctionName`, fully identifies the blueprint.
`State`	`FunctionInstanceState`	Current lifecycle state (see next subsection). Determines whether the instance can accept new requests.
`ContainerID`	`string`	Reference to the underlying container or microVM (e.g., Docker container ID, Firecracker VM ID).
`Host`	`string`	IP address or hostname where the instance is running. Used by the gateway to route HTTP requests.
`Port`	`int`	TCP port where the instance's HTTP server is listening. Combined with `Host` forms the endpoint.
`MemoryUsageMB`	`int`	Current memory usage in megabytes. Updated periodically from container stats for monitoring and scaling decisions.
`CPUUsage`	`float64`	Current CPU usage as percentage of allocated quota. Also from container stats.
`CreatedAt`	`time.Time`	When the instance was first created. Used to calculate age for cleanup of stale instances.
`LastUsedAt`	`time.Time`	When the instance last processed a request. Critical for determining idle time and scale-down decisions.
`TerminatedAt`	`*time.Time`	When the instance was terminated (nil if still running). Used for cleanup and accounting.
`InvocationCount`	`int`	Number of requests this instance has processed. Helps identify "hot" instances for sticky routing or debugging.
`TotalDurationMS`	`int64`	Cumulative milliseconds spent executing requests. Used for performance analysis and cost estimation.

> **Design Rationale:** Separating the blueprint (`FunctionDefinition`) from the running instance (`FunctionInstance`) allows multiple instances of the same function to run concurrently (for scaling) while sharing the same immutable specification. The instance tracks runtime-specific details like resource usage and lifecycle state.

InvocationRequest: The Work Item

An `InvocationRequest` represents a **single call to execute a function**. Think of it as a **customer order ticket** that arrives at the restaurant—it specifies what function to run, provides input data, and includes metadata about how to handle the request (deadlines, callbacks, etc.).

Field	Type	Description
`ID`	`string`	Unique identifier for this invocation (UUID). Used for idempotency, tracing, and result lookup.
`FunctionName`	`string`	Name of the function to invoke. The gateway uses this to route to the appropriate function pool.
`FunctionVersion`	`string`	Optional version constraint (e.g., "latest", "v1.2"). If empty, defaults to the latest active version.
`Payload`	`[]byte`	Raw request body (e.g., JSON, binary data) passed to the function as input.
`Synchronous`	`bool`	Whether the caller waits for a response (true) or the request is queued for async execution (false).
`Deadline`	`time.Time`	Absolute time by which the function must complete. Used for timeout enforcement and queue priority.
`RequestID`	`string`	Client-provided identifier for correlation (often passed through in headers). Distinct from system `ID`.
`Source`	`string`	Event source that triggered the invocation (e.g., "api-gateway", "s3:bucket-created", "cron:nightly").
`CreatedAt`	`time.Time`	When the request arrived at the system. Used for queue latency metrics.
`PreferredInstanceID`	`string`	Optional hint for sticky routing—if provided, the gateway tries to route to this specific instance.
`CallbackURL`	`string`	For async invocations, URL to POST the result to when complete. Empty for synchronous requests.

> **Key Insight:** The `InvocationRequest` captures both the **data** (`Payload`) and **metadata** (`Deadline`, `CallbackURL`) about an execution. This separation allows the system to handle scheduling, routing, and lifecycle management independently of the actual function logic.

Supporting Data Structures

Several supporting types enable detailed management and observability:

Container and ContainerManager Types These types abstract over the underlying container runtime (Docker, containerd, Firecracker):

Type/Field	Description
`Container`	Represents a managed container/microVM with its ID, image, status, labels, and network ports.
`CreateContainerRequest`	Configuration for creating a new isolated environment—image, command, environment variables, resource limits, security options, and port bindings.
`ContainerStats`	Resource usage metrics sampled from a running container: CPU %, memory usage/limit, network I/O, block I/O, and process count.
`ContainerManager`	Interface defining operations for managing isolated environments: create, start, stop, remove, execute commands, and collect stats.
`ListFilter`	Criteria for listing containers (by labels, status). Used to find function instances by function name/version.

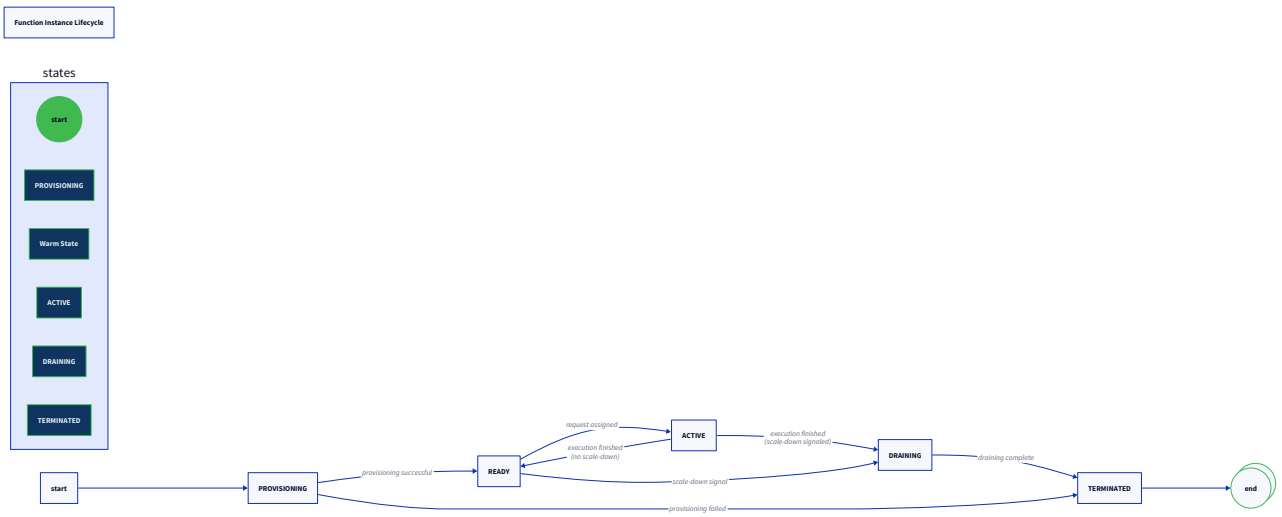
Invocation Tracking Types These types track the progress and outcome of function executions:

Type/Field	Description
`InvocationResponse`	Result returned from a function execution: output payload, duration, memory used, any error, and which instance processed it.
`InvocationStatus`	Tracks the state of an asynchronous invocation: pending, running, completed, failed, or timed out, with pointer to the result when available.

Metrics Type The `Metrics` struct aggregates all Prometheus metrics for observability: cold start latency histograms, invocation duration histograms, active invocation gauges, error counters categorized by function and error type, container operation counters, and resource usage gauges per instance.

Lifecycle State Definitions

A `FunctionInstance` transitions through a well-defined set of states during its lifetime. Understanding these states is critical for proper request routing, scaling decisions, and resource cleanup. The state transitions can be visualized as:



FunctionInstanceState Constants

Each state represents a specific phase in the instance's existence:

Constant	Description
`InstanceStateProvisioning`	The container is being created and started. The runtime environment is being prepared but is not yet ready to accept requests. This is the initial state after `CreateContainer`.
`InstanceStateWarm`	The container is running, the function runtime (e.g., Python interpreter, JVM) is initialized, dependencies are loaded, and the instance is **ready to immediately accept requests**. This is the state for instances in the warm pool.
`InstanceStateActive`	The instance is currently executing a request. No new requests should be routed to it until it completes. This state ensures per-instance concurrency limits are enforced.
`InstanceStateDraining`	The instance is finishing its current request(s) but will not accept new ones. Used during scale-down or before termination to gracefully handle in-flight requests.
`InstanceStateTerminated`	The container has been stopped and all resources cleaned up. This is a terminal state—the instance record may be kept for auditing but is no longer active.
`InstanceStateError`	The container has crashed, failed health checks, or entered an unrecoverable state. The system should replace it with a new instance and log the error for debugging.

State Transition Logic

The following table defines the events that trigger state changes and the actions the system must take during each transition:

Current State	Event	Next State	Action Taken
`Provisioning`	Container started successfully, runtime initialized	`Warm`	Add instance to warm pool; make it available for routing
`Provisioning`	Container startup failed (timeout, OOM, image pull error)	`Error`	Log error; notify controller to possibly retry creation
`Warm`	Request assigned to instance	`Active`	Mark instance as busy; start execution timeout timer
`Active`	Request completed (success or error)	`Warm`	Update `LastUsedAt`; return to warm pool if healthy
`Active`	Request timed out or instance crashed	`Error`	Force stop container; log failure; remove from pools
`Warm`	Scale-down signal received (instance selected for termination)	`Draining`	Remove from warm pool; allow current request to finish but don't assign new ones
`Warm`	Health check failed	`Error`	Mark unhealthy; schedule replacement instance
`Draining`	Current request completed (if any)	`Terminated`	Call `StopContainer` and `RemoveContainer`; clean up resources
`Draining`	Draining timeout exceeded (requests taking too long)	`Error`	Force terminate container; log warning about stuck requests
Any state except `Terminated`	System shutdown or forced termination	`Terminated`	Immediately stop container; skip graceful draining

> **Critical Design Principle:** State transitions must be **atomic and thread-safe**. Multiple components (gateway assigning requests, health checker, scaler) may concurrently attempt to modify an instance's state. Use a state machine with proper locking or compare-and-swap operations to prevent race conditions.

Container Status Mapping

The `FunctionInstanceState` is a higher-level abstraction than the underlying `ContainerStatus`. While they correlate, they serve different purposes:

- `ContainerStatus` (e.g., `StatusRunning`, `StatusExited`) reflects the raw container runtime state
- `FunctionInstanceState` reflects the business logic state of the instance within our serverless system

The typical mapping is:

- `ContainerStatus` = `StatusCreated` → `FunctionInstanceState` = `Provisioning`
- `ContainerStatus` = `StatusRunning` → `FunctionInstanceState` = `Warm` or `Active`
- `ContainerStatus` = `StatusExited` → `FunctionInstanceState` = `Terminated` or `Error`

> **Key Insight:** The `FunctionInstanceState` adds semantic meaning beyond the container's raw status. An instance in `Warm` state has a `StatusRunning` container, but not all `StatusRunning` containers are necessarily in `Warm` state—they could be in `Active` or `Draining`. This additional layer of abstraction allows the system to implement higher-level behaviors like warm pooling and graceful shutdown.

Implementation Guidance

> **Technology Note:** The data model is primarily implemented through Go structs with JSON tags for serialization. We'll use the `time.Time` type for timestamps (which serializes to RFC3339 format in JSON) and pointers for optional fields (like `TerminatedAt`).

Recommended File Structure

Organize the data model types logically by domain:

```
project-root/
├── internal/
│   ├── types/
│   │   ├── function.go          # FunctionDefinition, FunctionInstance, FunctionInstanceState
│   │   ├── invocation.go        # InvocationRequest, InvocationResponse, InvocationStatus
│   │   ├── container.go         # Container, CreateContainerRequest, ContainerStats, ListFilter
│   │   ├── metrics.go           # Metrics struct and constructor
│   │   └── constants.go         # All constants (InstanceState*, ContainerStatus*, etc.)
│   ├── container/
│   │   └── manager.go           # ContainerManager interface and implementations
│   └── api/
│       └── models.go            # Request/Response structs for HTTP API (if different)
└── pkg/
    └── timeutil/                # Optional: time utilities for deadline calculations
```

Core Data Structure Implementation

Here's starter code for the key data structures with proper JSON serialization:

```go
// internal/types/function.go

package types


import (

        "time"

)


// FunctionDefinition represents the immutable blueprint for a function

type FunctionDefinition struct {

        Name         string            `json:"name"`

        Runtime      string            `json:"runtime"`

        Handler      string            `json:"handler"`

        CodePath     string            `json:"codePath"`

        Timeout      int               `json:"timeout"`      // seconds

        MemoryMB     int               `json:"memoryMB"`

        CPUQuota     int64             `json:"cpuQuota"`     // percentage of a core (100 = 1 core)

        EnvVars      map[string]string `json:"envVars"`

        Dependencies []string          `json:"dependencies"`

}


// Validate checks if the FunctionDefinition has required fields

func (fd *FunctionDefinition) Validate() error {

        // TODO 1: Check that Name is non-empty and matches allowed pattern (e.g., no spaces)

        // TODO 2: Verify Runtime is supported (e.g., "python3.9", "go1.19", "nodejs18")

        // TODO 3: Ensure Handler is non-empty

        // TODO 4: Validate Timeout is between 1 and maximum allowed (e.g., 900 seconds)

        // TODO 5: Validate MemoryMB is between minimum (e.g., 128) and maximum allowed

        // TODO 6: Verify CPUQuota is positive and within limits

        // TODO 7: Check CodePath is a valid URI format

        return nil

}


// FunctionInstanceState represents the lifecycle state of a FunctionInstance

type FunctionInstanceState string


const (

        InstanceStateProvisioning FunctionInstanceState = "PROVISIONING"

        InstanceStateWarm         FunctionInstanceState = "WARM"

        InstanceStateActive       FunctionInstanceState = "ACTIVE"

        InstanceStateDraining     FunctionInstanceState = "DRAINING"

        InstanceStateTerminated   FunctionInstanceState = "TERMINATED"

        InstanceStateError        FunctionInstanceState = "ERROR"

)
```

```go
// FunctionInstance represents a running instance of a function
type FunctionInstance struct {
	ID              string              `json:"id"`
	FunctionName    string              `json:"functionName"`
	FunctionVersion string              `json:"functionVersion"`
	State           FunctionInstanceState `json:"state"`
	ContainerID     string              `json:"containerId"`
	Host            string              `json:"host"`
	Port            int                 `json:"port"`
	MemoryUsageMB   int                 `json:"memoryUsageMB"`
	CPUUsage        float64             `json:"cpuUsage"`
	CreatedAt       time.Time           `json:"createdAt"`
	LastUsedAt      time.Time           `json:"lastUsedAt"`
	TerminatedAt    *time.Time          `json:"terminatedAt,omitempty"` // Pointer for optional field
	InvocationCount int                 `json:"invocationCount"`
	TotalDurationMS int64               `json:"totalDurationMs"`
}

// IsActive returns true if the instance can accept new requests
func (fi *FunctionInstance) IsActive() bool {
	// TODO 1: Return true only if State is InstanceStateWarm
	// TODO 2: Consider also checking that ContainerID is not empty
	// TODO 3: Consider adding a health check timestamp to avoid routing to stale instances
	return false
}

// Age returns how long the instance has been running
func (fi *FunctionInstance) Age() time.Duration {
	// TODO 1: If TerminatedAt is set, return TerminatedAt.Sub(CreatedAt)
	// TODO 2: Otherwise, return time.Since(CreatedAt)
	return 0
}

// IdleDuration returns how long the instance has been idle
func (fi *FunctionInstance) IdleDuration() time.Duration {
	// TODO 1: Return time.Since(LastUsedAt)
	// TODO 2: Handle edge case where LastUsedAt is zero time (never used)
	return 0
}
```

```go
// internal/types/invocation.go

package types


import (

	"time"

)


// InvocationRequest represents a single function execution request

type InvocationRequest struct {

	ID                 string    `json:"id"`

	FunctionName       string    `json:"functionName"`

	FunctionVersion    string    `json:"functionVersion,omitempty"`

	Payload            []byte    `json:"payload"`

	Synchronous        bool      `json:"synchronous"`

	Deadline           time.Time `json:"deadline"`

	RequestID          string    `json:"requestId,omitempty"`

	Source             string    `json:"source,omitempty"`

	CreatedAt          time.Time `json:"createdAt"`

	PreferredInstanceID string    `json:"preferredInstanceId,omitempty"`

	CallbackURL        string    `json:"callbackUrl,omitempty"`

}


// IsExpired returns true if the request has passed its deadline

func (ir *InvocationRequest) IsExpired() bool {

	// TODO 1: Return true if time.Now().After(Deadline)

	// TODO 2: Consider adding a small grace period (e.g., 100ms) for clock skew

	return false

}


// TimeRemaining returns how much time is left before the deadline

func (ir *InvocationRequest) TimeRemaining() time.Duration {

	// TODO 1: Return Deadline.Sub(time.Now())

	// TODO 2: Clamp to minimum of 0 (no negative durations)

	return 0

}


// InvocationStatusType represents the state of an asynchronous invocation

type InvocationStatusType string


const (

	InvocationStatusPending   InvocationStatusType = "PENDING"

	InvocationStatusRunning   InvocationStatusType = "RUNNING"

	InvocationStatusCompleted InvocationStatusType = "COMPLETED"
```

```
        InvocationStatusFailed    InvocationStatusType = "FAILED"
        InvocationStatusTimeout   InvocationStatusType = "TIMEOUT"
)

// InvocationStatus tracks the progress of an asynchronous invocation
type InvocationStatus struct {
        RequestID    string               `json:"requestId"`
        Status       InvocationStatusType `json:"status"`
        Result       *InvocationResponse  `json:"result,omitempty"`
        LastUpdated  time.Time            `json:"lastUpdated"`
}

// InvocationResponse contains the result of a function execution
type InvocationResponse struct {
        RequestID      string    `json:"requestId"`
        Payload        []byte    `json:"payload,omitempty"`
        DurationMS     int64     `json:"durationMs"`
        MemoryUsedMB   int       `json:"memoryUsedMb"`
        BilledDurationMS int64   `json:"billedDurationMs"` // Rounded up to nearest billing increment
        Error          string    `json:"error,omitempty"`
        ErrorType      string    `json:"errorType,omitempty"` // e.g., "UserError", "RuntimeError"
        InstanceID     string    `json:"instanceId"`
        StartedAt      time.Time `json:"startedAt"`
        FinishedAt     time.Time `json:"finishedAt"`
}
```

```
// internal/types/constants.go
package types

// ContainerStatus represents the state of a container in the container runtime
type ContainerStatus string

const (
        StatusCreated    ContainerStatus = "created"
        StatusRunning    ContainerStatus = "running"
        StatusPaused     ContainerStatus = "paused"
        StatusRestarting ContainerStatus = "restarting"
        StatusExited     ContainerStatus = "exited"
        StatusDead       ContainerStatus = "dead"
)
```
GO

Data Model Validation Helper

Add a simple validation utility to ensure data integrity:

```go
// internal/types/validation.go

package types


import (

        "errors"

        "net/url"

        "regexp"

)


var (

        ErrInvalidFunctionName = errors.New("function name must contain only alphanumeric characters and hyphens")

        ErrInvalidRuntime      = errors.New("unsupported runtime")

        ErrInvalidTimeout      = errors.New("timeout must be between 1 and 900 seconds")

        ErrInvalidMemory       = errors.New("memory must be between 128 and 10240 MB")

)


// ValidateFunctionDefinition performs comprehensive validation

func ValidateFunctionDefinition(fd *FunctionDefinition) error {

        // Name validation

        nameRegex := regexp.MustCompile(`^[a-zA-Z0-9\-]+$`)

        if !nameRegex.MatchString(fd.Name) {

                return ErrInvalidFunctionName

        }


        // Runtime validation

        supportedRuntimes := map[string]bool{

                "python3.9": true,

                "go1.19":    true,

                "nodejs18":  true,

        }

        if !supportedRuntimes[fd.Runtime] {

                return ErrInvalidRuntime

        }


        // Timeout validation

        if fd.Timeout < 1 || fd.Timeout > 900 {

                return ErrInvalidTimeout

        }


        // Memory validation

        if fd.MemoryMB < 128 || fd.MemoryMB > 10240 {

                return ErrInvalidMemory

        }
```

```
        // CodePath validation
        if _, err := url.Parse(fd.CodePath); err != nil {
                return errors.New("invalid CodePath URL")
        }

        // Handler must be non-empty
        if fd.Handler == "" {
                return errors.New("handler cannot be empty")
        }

        return nil
}
```

Language-Specific Tips for Go Implementation

1. **Use `time.Time` for timestamps**: It serializes well to JSON (RFC3339) and has rich methods for time calculations.
2. **Use pointer fields for optional values**: Fields like `TerminatedAt *time.Time` will be omitted from JSON when nil.
3. **Validate early**: Implement `Validate()` methods and call them before persisting or using data structures.
4. **Make state transitions atomic**: Use `sync.Mutex` or `sync.RWMutex` in the struct that manages `FunctionInstance` states to prevent race conditions.
5. **Consider using UUIDs for IDs**: Use `github.com/google/uuid` to generate unique identifiers that are globally unique and not guessable.

Milestone Checkpoint: Data Model Validation

After implementing the data structures, verify they work correctly:

```bash
# Run unit tests for the types package
go test ./internal/types/... -v

# Expected output should show tests passing for:
# - FunctionDefinition validation (valid and invalid cases)
# - FunctionInstance state methods (IsActive, Age, IdleDuration)
# - InvocationRequest deadline methods (IsExpired, TimeRemaining)
```

Create a simple test to verify serialization:

```go
// internal/types/function_test.go (example test)

func TestFunctionDefinition_JSONRoundTrip(t *testing.T) {

        fd := &FunctionDefinition{

                Name:     "test-function",

                Runtime:  "python3.9",

                Handler:  "handler.main",

                CodePath: "s3://bucket/test-v1.zip",

                Timeout:  30,

                MemoryMB: 256,

                CPUQuota: 100,

                EnvVars:  map[string]string{"KEY": "VALUE"},

        }



        // Marshal to JSON

        jsonData, err := json.Marshal(fd)

        assert.NoError(t, err)



        // Unmarshal back

        var fd2 FunctionDefinition

        err = json.Unmarshal(jsonData, &fd2)

        assert.NoError(t, err)



        // Should be equal

        assert.Equal(t, fd.Name, fd2.Name)

        assert.Equal(t, fd.Runtime, fd2.Runtime)

}
```

Component Design: Function Packaging

> **Milestone(s):** Milestone 1

Mental Model: The Shipping Dock and Warehouse

Imagine you're managing a massive shipping facility where customers send you boxes of parts and instructions to assemble specialized machines. Each customer's box contains the blueprints (source code), a list of required components (dependencies), and assembly instructions (build configuration). Your job is to:

1. **Receive shipments** at the dock (upload API)
2. **Inspect and validate** the contents (package validation)
3. **Assemble complete machines** by gathering all required parts (dependency resolution)
4. **Test the assembly** to ensure it works (build validation)
5. **Store finished machines** in labeled warehouse bins with unique serial numbers (content-addressable storage)
6. **Track revisions** so you can always retrieve the exact same machine later (immutable versioning)

Just as a warehouse worker doesn't need to know how to operate every machine, the runtime doesn't need to understand the function's internal logic—it just needs to retrieve the pre-assembled package and plug it into an execution environment. This separation of concerns allows the packaging subsystem to focus on reliable, reproducible builds while the execution subsystem focuses on fast, secure runtime.

The critical insight is that **packaging happens once per deployment, but execution happens thousands or millions of times**. Therefore, we invest more computational resources during packaging (thorough dependency resolution, optimization, validation) to make execution as lightweight and predictable as possible.

Interface and Workflow

The packaging subsystem exposes two primary interfaces: an external HTTP API for users to upload functions, and internal interfaces for language-specific bundlers to process those functions.

External API: Function Upload and Management

Users interact with the packaging system through a RESTful HTTP API that supports:

Endpoint	Method	Request Body	Response	Description
`/v1/functions`	`POST`	Multipart form with code archive, `function.yaml`	`201 Created` with `{"function_id": "...", "version": "..."}`	Upload new function version
`/v1/functions/{name}/versions`	`GET`	None	`200 OK` with list of versions	List all versions of a function
`/v1/functions/{name}/versions/{version}`	`GET`	None	`200 OK` with function metadata	Get specific version details
`/v1/functions/{name}/versions/{version}`	`DELETE`	None	`204 No Content`	Delete specific version (if not in use)

The `function.yaml` configuration file follows this schema:

Field	Type	Required	Description
`name`	string	Yes	Unique function name (slug format: `[a-z0-9-]+`)
`runtime`	string	Yes	One of: `go1.x`, `python3.9`, `nodejs18.x`, `java11`
`handler`	string	Yes	Entry point format varies by runtime: - Go: `package.FunctionName` - Python: `module.function` - Node.js: `module.exportName` - Java: `com.example.Handler::handleRequest`
`timeout`	integer	No	Maximum execution time in seconds (default: 30)
`memory`	integer	No	Memory allocation in MB (default: 128)
`environment`	map[string]string	No	Environment variables injected at runtime
`dependencies`	array of strings	No	Language-specific dependency specifiers

Internal Workflow: From Upload to Stored Artifact

When a function upload request arrives, the system executes this 10-step workflow:

1. **Request Validation**: Verify the multipart form contains both `function.yaml` and at least one code file (ZIP, TAR, or single file).
2. **Configuration Parsing**: Parse `function.yaml` into a `FunctionDefinition` struct, validating all required fields and runtime compatibility.
3. **Temporary Storage**: Extract uploaded files to a temporary working directory with a unique session ID.
4. **Language Detection**: Based on the `runtime` field, route to the appropriate language bundler.
5. **Dependency Resolution**: The language bundler analyzes the code to identify all required dependencies:
   - For interpreted languages (Python, Node.js): Parse dependency manifests (`requirements.txt`, `package.json`) and resolve transitive dependencies, handling version conflicts.
   - For compiled languages (Go, Java): Compile the code, capturing all imported packages and libraries.
6. **Build Process**: Create a complete, self-contained artifact:
   - **Python**: Create a virtual environment with all dependencies, then package into a ZIP with the function code.
   - **Node.js**: Run `npm install --production` and bundle into a single directory.
   - **Go**: Compile to a standalone binary for the target architecture.
   - **Java**: Compile to a JAR with all dependencies shaded/fat-jarred.
7. **Artifact Optimization**: Apply optimizations to reduce package size:
   - Remove development dependencies, documentation, test files.

- Compress resources (minify JavaScript, strip debug symbols from binaries).
- Deduplicate common dependencies across functions (layer sharing).

8. **Integrity Check**: Compute SHA-256 hash of the final artifact for content addressing.
9. **Storage**: Store the artifact in the content-addressable storage backend using the hash as the key.
10. **Metadata Registration**: Record the function version metadata in the registry, linking the function name and version to the storage hash.

Internal Interfaces for Language Bundlers

The packaging subsystem defines this interface for language-specific implementations:

Method	Parameters	Returns	Description
`ResolveDependencies`	`ctx context.Context, workDir string, deps []string`	`([]string, error)`	Analyze code and return complete dependency list
`BuildArtifact`	`ctx context.Context, workDir string, def FunctionDefinition`	`(string, error)`	Build self-contained artifact, return path
`ValidateHandler`	`ctx context.Context, workDir string, def FunctionDefinition`	`error`	Verify handler exists and is callable
`GetBaseImage`	`runtime string`	`string`	Return container image name for this runtime

Each language runtime (Go, Python, Node.js, Java) provides its own implementation of this interface. The packaging coordinator calls these methods in sequence, handling errors and cleanup between steps.

ADR: Artifact Storage Strategy

> **Decision: Use Content-Addressable Storage with Registry Index**
>
> **Context**: We need to store function artifacts durably and retrieve them efficiently. Artifacts range from small (Go binaries ~5MB) to large (Java applications with dependencies ~100MB+). We must support:
>
> - Immutable versioning (once stored, never modified)
> - Efficient deduplication (identical dependencies across functions)
> - Fast retrieval by both content hash and function name/version
> - Scalability to thousands of functions with multiple versions each
>
> **Options Considered**:
>
> 1. **Object Storage with Hierarchical Keys** (e.g., S3 with `{function}/{version}/artifact.zip`)
> 2. **Content-Addressable Storage** (e.g., store by SHA-256 hash, maintain separate index)
> 3. **Container Registry** (e.g., Docker registry, store as OCI images)
>
> **Decision**: Use content-addressable storage (option 2) with a separate registry database indexing function names and versions to content hashes.
>
> **Rationale**:
>
> - **Deduplication**: Identical artifacts (common dependencies, same code) automatically deduplicate because they have the same hash. This saves significant storage for common runtimes and libraries.
> - **Immutability**: Content addressing guarantees immutability—the hash is the content's fingerprint. Any tampering changes the hash, making detection trivial.
> - **Cache Efficiency**: CDNs and caching layers can use the hash as a perfect cache key. Two functions with identical dependencies can share cached layers.
> - **Simplified Garbage Collection**: We can implement reference counting on hashes—delete artifacts only when no function version references them.
> - **Flexibility**: We can store artifacts in any backing store (S3, filesystem, IPFS) since the abstraction depends only on hash-based retrieval.
>
> **Consequences**:
>
> - **Added Complexity**: Need a separate registry database to map `(function, version) → hash`. Two-phase writes (store artifact, then update registry).
> - **Slower Lookup**: To get an artifact by function name, we must first query the registry for the hash, then fetch by hash (two operations).
> - **No Native Versioning**: The storage layer doesn't understand semantic versioning—that's managed by the registry.

Option	Pros	Cons	Why Not Chosen
Object Storage with Hierarchical Keys	Simple, direct mapping, S3 versioning available	No automatic deduplication, harder cache optimization, mutable without proper discipline	Lacks automatic deduplication which is critical for common dependencies like AWS SDK
Content-Addressable Storage	**Automatic deduplication, immutable by design, cache-friendly, storage-agnostic**	**Requires separate index, two-step retrieval**	**Chosen for deduplication and immutability benefits**
Container Registry (OCI)	Standard format, tooling ecosystem, layer sharing	Heavyweight for simple functions, complex to implement, overkill for non-container execution	Too complex for our needs; we're not running containers directly but might use them as base images

Common Pitfalls in Function Packaging

⚠️ Pitfall: Dependency Hell and Version Conflicts

Description: When function A requires `library-v1.2` and function B requires `library-v2.0`, but they share a common base environment, one function will break. Transitive dependencies compound this—`library-v1.2` might require `dep-alpha-v3.0` while `library-v2.0` requires `dep-alpha-v4.0`, which are incompatible.

Why It's Wrong: This causes non-deterministic behavior. A function that works during packaging might fail at runtime if a different function's deployment changes the shared environment. It violates the principle of isolation between functions.

How to Fix:

- Use **per-function virtual environments** (Python), `node_modules` isolation (Node.js), or shaded JARs (Java).
- For compiled languages (Go), statically link dependencies into the binary.
- Never share mutable dependency directories between functions.
- Implement **dependency conflict detection** during packaging—fail the build if incompatible transitive dependencies are detected.

⚠️ Pitfall: Bloated Packages from Development Artifacts

Description: Including `node_modules` with dev dependencies, Python virtual environments with `.pyc` files, Java JARs with source code and documentation, or Go binaries with debug symbols. A simple 50-line function can become a 300MB package.

Why It's Wrong: Large packages increase cold start time (more data to transfer), consume more storage, and waste memory. They also expose potentially sensitive information (source code, comments, debug symbols).

How to Fix:

- Implement **production-only packaging**: Exclude dev dependencies, documentation, tests, examples.
- Apply **aggressive compression**: Use ZIP with maximum compression, strip debug symbols from binaries.
- **Deduplicate common layers**: Store common runtime dependencies (like AWS SDK) as shared layers that can be mounted read-only.
- Set **maximum package size limits** (e.g., 250MB) and reject oversized packages.

⚠️ Pitfall: Improper or Missing Versioning

Description: Using mutable tags like `latest`, auto-incrementing version numbers without consistency, or allowing package modification after upload.

Why It's Wrong: Breaks reproducibility and rollback capabilities. If a function starts failing, you cannot reliably revert to the previous working version. Also prevents deterministic deployments across environments (dev, staging, prod).

How to Fix:

- Use **immutable, content-based versioning**: The artifact hash (SHA-256) becomes part of the version identifier.
- Support **semantic versioning** as user-friendly aliases: `myfunction:v1.2.3` points to hash `abc123...`.
- Implement **version immutability guarantee**: Once stored, an artifact cannot be modified or deleted while referenced.
- Maintain **version lineage**: Track which version succeeded which, enabling rollback graphs.

⚠️ Pitfall: Incomplete Validation Leading to Runtime Failures

Description: Only validating the package structure but not testing if the function actually works. The handler might not exist, dependencies might be missing at runtime, or the function signature might be wrong.

Why It's Wrong: Defers discovery of problems until invocation time, which is worse for users (harder to debug) and the system (wastes resources provisioning environments for broken functions).

How to Fix:

- **Validate handler existence**: For interpreted languages, parse the code to ensure the handler module/function exists.
- **Test compilation**: For compiled languages, actually compile the code during packaging.

- **Optional smoke test**: Provide a flag to run a test invocation during packaging with a sample payload.
- **Runtime compatibility check**: Verify the function works with the specific runtime version (Python 3.9 vs 3.10 differences).

Implementation Guidance for Packaging

Technology Recommendations

Component	Simple Option	Advanced Option
Upload API	HTTP multipart form with `net/http`	gRPC with chunked streaming for large packages
Dependency Resolution	Language-native tools (`go mod` , `npm` , `pip`)	Custom resolver with vulnerability scanning
Artifact Storage	Local filesystem with hash-based directories	S3/MinIO with content addressing
Registry Database	SQLite (embedded)	PostgreSQL with connection pooling
Build Environment	Local OS with language runtimes installed	Docker containers for hermetic builds

Recommended File/Module Structure

```
serverless-runtime/
├── cmd/
│   ├── packaging-server/          # Standalone packaging service
│   │   └── main.go
│   └── gateway/                   # Main gateway (includes packaging routes)
│       └── main.go
├── internal/
│   ├── packaging/
│   │   ├── api/                   # HTTP/API handlers
│   │   │   ├── upload.go
│   │   │   ├── versions.go
│   │   │   └── middleware.go
│   │   ├── bundler/               # Language-specific bundlers
│   │   │   ├── interface.go
│   │   │   ├── go_bundler.go
│   │   │   ├── python_bundler.go
│   │   │   ├── nodejs_bundler.go
│   │   │   └── java_bundler.go
│   │   ├── registry/              # Function version registry
│   │   │   ├── registry.go
│   │   │   ├── sqlite_store.go
│   │   │   └── postgres_store.go
│   │   ├── storage/               # Artifact storage backends
│   │   │   ├── interface.go
│   │   │   ├── filesystem_store.go
│   │   │   └── s3_store.go
│   │   ├── types/                 # Packaging-specific types
│   │   │   ├── function_def.go
│   │   │   └── artifact.go
│   │   └── coordinator.go         # Orchestrates the packaging workflow
│   └── function/                  # Shared function types (from Data Model)
│       └── types.go
└── pkg/
    └── compression/               # Reusable compression utilities
        ├── zip.go
        └── tar.go
```

Infrastructure Starter Code: Upload API Handler

```go
// internal/packaging/api/upload.go

package api

import (
    "context"
    "crypto/sha256"
    "encoding/hex"
    "fmt"
    "io"
    "mime/multipart"
    "net/http"
    "os"
    "path/filepath"
    "time"

    "github.com/your-org/serverless-runtime/internal/packaging"
    "github.com/your-org/serverless-runtime/internal/packaging/types"
)

// UploadHandler handles multipart form uploads for function packages
type UploadHandler struct {
    coordinator *packaging.Coordinator
    maxUploadMB int64
    tempDir     string
}

// NewUploadHandler creates a new upload handler with configuration
func NewUploadHandler(coordinator *packaging.Coordinator, maxUploadMB int64) *UploadHandler {
    return &UploadHandler{
        coordinator: coordinator,
        maxUploadMB: maxUploadMB,
        tempDir:     os.TempDir(),
    }
}

// ServeHTTP implements http.Handler
func (h *UploadHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    // 1. Limit request size
    r.Body = http.MaxBytesReader(w, r.Body, h.maxUploadMB*1024*1024)

    // 2. Parse multipart form
    if err := r.ParseMultipartForm(10 << 20); // 10MB in memory, rest on disk
```

```
err != nil {
    http.Error(w, "Failed to parse multipart form: "+err.Error(), http.StatusBadRequest)
    return
}

// 3. Extract function.yaml
funcFile, funcHeader, err := r.FormFile("function.yaml")
if err != nil {
    http.Error(w, "Missing required file: function.yaml", http.StatusBadRequest)
    return
}
defer funcFile.Close()

// 4. Extract code archive
codeFile, codeHeader, err := r.FormFile("code")
if err != nil {
    http.Error(w, "Missing required file: code", http.StatusBadRequest)
    return
}
defer codeFile.Close()

// 5. Create temporary workspace
workspace, err := h.createWorkspace()
if err != nil {
    http.Error(w, "Failed to create workspace: "+err.Error(), http.StatusInternalServerError)
    return
}
defer os.RemoveAll(workspace) // Cleanup

// 6. Save uploaded files to workspace
funcDefPath, err := h.saveUploadedFile(funcFile, funcHeader, workspace, "function.yaml")
if err != nil {
    http.Error(w, "Failed to save function definition: "+err.Error(), http.StatusInternalServerError)
    return
}

codePath, err := h.saveUploadedFile(codeFile, codeHeader, workspace, "code")
if err != nil {
    http.Error(w, "Failed to save code: "+err.Error(), http.StatusInternalServerError)
    return
```

```
    }

    // 7. Parse and validate function definition
    funcDef, err := h.parseFunctionDefinition(funcDefPath)
    if err != nil {
        http.Error(w, "Invalid function definition: "+err.Error(), http.StatusBadRequest)
        return
    }

    // 8. Extract code archive if it's a ZIP/TAR
    if err := h.extractArchive(codePath, workspace); err != nil {
        http.Error(w, "Failed to extract code archive: "+err.Error(), http.StatusBadRequest)
        return
    }

    // 9. Process through coordinator
    ctx, cancel := context.WithTimeout(r.Context(), 5*time.Minute)
    defer cancel()

    result, err := h.coordinator.ProcessUpload(ctx, funcDef, workspace)
    if err != nil {
        http.Error(w, "Packaging failed: "+err.Error(), http.StatusInternalServerError)
        return
    }

    // 10. Return success response
    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(http.StatusCreated)
    fmt.Fprintf(w, `{"function_id": "%s", "version": "%s", "artifact_hash": "%s"}`,
        result.FunctionName, result.Version, result.ArtifactHash)
}

// Helper method to create unique workspace directory
func (h *UploadHandler) createWorkspace() (string, error) {
    dir := filepath.Join(h.tempDir, fmt.Sprintf("func-pkg-%d", time.Now().UnixNano()))
    if err := os.MkdirAll(dir, 0755); err != nil {
        return "", fmt.Errorf("mkdir failed: %w", err)
    }
    return dir, nil
}
```

```go
// Helper method to save uploaded file

func (h *UploadHandler) saveUploadedFile(file multipart.File, header *multipart.FileHeader, workspace, destName string) (string, error) {

    destPath := filepath.Join(workspace, destName)

    dest, err := os.Create(destPath)

    if err != nil {

        return "", fmt.Errorf("create file failed: %w", err)

    }

    defer dest.Close()



    if _, err := io.Copy(dest, file); err != nil {

        return "", fmt.Errorf("copy failed: %w", err)

    }



    return destPath, nil

}


// Helper method to compute file hash

func computeFileHash(path string) (string, error) {

    f, err := os.Open(path)

    if err != nil {

        return "", err

    }

    defer f.Close()



    hasher := sha256.New()

    if _, err := io.Copy(hasher, f); err != nil {

        return "", err

    }



    return hex.EncodeToString(hasher.Sum(nil)), nil

}
```

Core Logic Skeleton: Packaging Coordinator

```go
// internal/packaging/coordinator.go

package packaging

import (
    "context"
    "fmt"
    "os"
    "path/filepath"

    "github.com/your-org/serverless-runtime/internal/packaging/bundler"
    "github.com/your-org/serverless-runtime/internal/packaging/registry"
    "github.com/your-org/serverless-runtime/internal/packaging/storage"
    "github.com/your-org/serverless-runtime/internal/packaging/types"
)

// Coordinator orchestrates the packaging workflow
type Coordinator struct {
    registry    registry.Registry
    storage     storage.ArtifactStorage
    bundlers    map[string]bundler.Bundler // keyed by runtime
    workspaceDir string
}

// ProcessUploadResult contains results of a successful packaging operation
type ProcessUploadResult struct {
    FunctionName  string
    Version       string
    ArtifactHash  string
    ArtifactSize  int64
    Dependencies  []string
}

// ProcessUpload executes the complete packaging workflow
func (c *Coordinator) ProcessUpload(ctx context.Context, def *types.FunctionDefinition, sourceDir string) (*ProcessUploadResult, error) {
    // TODO 1: Validate the FunctionDefinition (check required fields, runtime support)
    // Hint: Use def.Validate() method

    // TODO 2: Select appropriate bundler based on def.Runtime
    // Hint: Look up in c.bundlers map, return error if runtime not supported

    // TODO 3: Validate the handler exists in source code
    // Hint: Call bundler.ValidateHandler(ctx, sourceDir, def)
```

```
    // TODO 4: Resolve dependencies
    // Hint: Call bundler.ResolveDependencies(ctx, sourceDir, def.Dependencies)
    // Store the complete dependency list for metadata

    // TODO 5: Create temporary build directory
    // Hint: Use os.MkdirTemp with c.workspaceDir as parent

    // TODO 6: Build artifact
    // Hint: Call bundler.BuildArtifact(ctx, sourceDir, def)
    // The bundler returns path to the built artifact file

    // TODO 7: Compute artifact hash (SHA-256)
    // Hint: Read the artifact file and compute hash using sha256.Sum256
    // Convert to hex string for storage key

    // TODO 8: Check if artifact already exists in storage (deduplication)
    // Hint: Call c.storage.Exists(ctx, hash)
    // If exists, skip to step 10

    // TODO 9: Store artifact in content-addressable storage
    // Hint: Call c.storage.Store(ctx, hash, artifactPath)
    // Get artifact size for metadata

    // TODO 10: Register function version in registry
    // Hint: Create registry.FunctionVersion record with:
    // - FunctionName: def.Name
    // - Version: generateVersion(def, hash)
    // - ArtifactHash: hash
    // - Metadata: size, dependencies, creation time

    // TODO 11: Clean up temporary build directory
    // Hint: os.RemoveAll on build directory

    // TODO 12: Return ProcessUploadResult with all metadata
    return nil, fmt.Errorf("not implemented")
}

// GetBundler returns the bundler for a runtime
func (c *Coordinator) GetBundler(runtime string) (bundler.Bundler, error) {
    b, ok := c.bundlers[runtime]
```

```
    if !ok {
        return nil, fmt.Errorf("unsupported runtime: %s", runtime)
    }
    return b, nil
}

// generateVersion creates a version identifier from function definition and hash
func generateVersion(def *types.FunctionDefinition, hash string) string {
    // Use short hash (first 12 chars) for readability
    shortHash := hash[:12]
    return fmt.Sprintf("%s-%s", def.Name, shortHash)
}
```

Core Logic Skeleton: Go Language Bundler

```go
// internal/packaging/bundler/go_bundler.go

package bundler

import (
    "context"
    "fmt"
    "os"
    "os/exec"
    "path/filepath"
    "strings"

    "github.com/your-org/serverless-runtime/internal/packaging/types"
)

// GoBundler implements Bundler interface for Go functions
type GoBundler struct {
    goPath      string // Path to go binary
    buildTarget string // Target OS/architecture, e.g., "linux/amd64"
}

// NewGoBundler creates a new Go bundler
func NewGoBundler() (*GoBundler, error) {
    goPath, err := exec.LookPath("go")
    if err != nil {
        return nil, fmt.Errorf("go not found in PATH: %w", err)
    }

    return &GoBundler{
        goPath:      goPath,
        buildTarget: "linux/amd64", // Standard serverless target
    }, nil
}

// ResolveDependencies resolves Go dependencies using go mod
func (b *GoBundler) ResolveDependencies(ctx context.Context, workDir string, deps []string) ([]string, error) {
    // TODO 1: Check if go.mod exists in workDir
    // Hint: Use os.Stat on filepath.Join(workDir, "go.mod")
    // If not exists, create a minimal go.mod with module name

    // TODO 2: Run 'go mod tidy' to ensure go.mod and go.sync are consistent
    // Hint: Use exec.CommandContext with b.goPath, "mod", "tidy"
    // Set Dir to workDir, capture output
```

```
    // TODO 3: If deps provided, add them to go.mod
    // Hint: For each dep in deps, run 'go get <dep>'

    // TODO 4: Run 'go list -m all' to get complete dependency list
    // Hint: Parse output to get all module paths and versions

    // TODO 5: Return list of dependencies in format "module@version"
    return []string{}, fmt.Errorf("not implemented")
}

// BuildArtifact builds a standalone Go binary
func (b *GoBundler) BuildArtifact(ctx context.Context, workDir string, def types.FunctionDefinition) (string, error) {
    // TODO 1: Parse handler to extract package and function name
    // Go handlers format: "package.FunctionName"
    // Hint: Split on last "." - everything before is import path, after is function

    // TODO 2: Create temporary output directory for the binary
    // Hint: os.MkdirTemp

    // TODO 3: Build command: go build -o <output> <build flags>
    // Required flags:
    // - -ldflags="-s -w" // Strip debug symbols
    // - -trimpath        // Remove file system paths
    // - GOOS=linux, GOARCH=amd64 environment variables

    // TODO 4: Execute build command
    // Hint: Use exec.CommandContext, set Dir to workDir
    // Capture stdout/stderr for error reporting

    // TODO 5: Verify binary was created and is executable
    // Hint: os.Stat on output path, check file mode

    // TODO 6: Return path to built binary
    return "", fmt.Errorf("not implemented")
}

// ValidateHandler verifies the handler exists in Go code
func (b *GoBundler) ValidateHandler(ctx context.Context, workDir string, def types.FunctionDefinition) error {
    // TODO 1: Parse handler string format: "package.FunctionName"
    // Hint: strings.Split for last "."
```

```
    // TODO 2: Check if package directory exists in workDir
    // For relative packages (starting with "./"), check relative to workDir
    // For absolute packages, they should be in GOPATH/module cache

    // TODO 3: Use 'go list' to verify function exists in package
    // Command: go list -f '{{.Name}} {{.Doc}}' <package>
    // Parse output to find function

    // TODO 4: Check function signature matches expected handler format
    // Serverless Go handlers should be: func Handler(ctx context.Context, payload []byte) ([]byte, error)
    // We can't fully validate without compilation, but we can check via go/ast parsing
    // Simple approach: grep for function definition in source files

    return fmt.Errorf("not implemented")
}

// GetBaseImage returns container image for Go runtime
func (b *GoBundler) GetBaseImage(runtime string) string {
    // Use lightweight Alpine-based image for Go
    return "golang:alpine"
}
```

Language-Specific Hints for Go Implementation

- **Dependency Management**: Use `go mod` commands programmatically via `exec.Command`. Set `GOPROXY=direct` to avoid proxy caching during builds.
- **Cross-Compilation**: Set `GOOS=linux` and `GOARCH=amd64` environment variables when building for the serverless environment.
- **Binary Optimization**: Use `-ldflags="-s -w"` to strip debug symbols and reduce binary size by 20-30%.
- **Handler Validation**: For thorough validation, use the `go/ast` package to parse source files and verify function signatures match `func(context.Context, []byte) ([]byte, error)`.
- **Temporary Files**: Use `os.MkdirTemp` with cleanup deferred. For large builds, ensure adequate disk space in the temporary directory.

Milestone Checkpoint: Verifying Function Packaging

After implementing the packaging subsystem, verify it works:

1. **Start the packaging server**:

```bash
go run cmd/packaging-server/main.go --port 8080 --storage-dir /tmp/func-storage
```

2. **Create a test function**:

```bash
mkdir test-function

cat > test-function/function.yaml << 'EOF'

name: hello-world

runtime: go1.x

handler: main.Handler

timeout: 30

memory: 128

EOF


cat > test-function/main.go << 'EOF'

package main


import (

    "context"

    "fmt"

)


func Handler(ctx context.Context, input []byte) ([]byte, error) {

    return []byte(fmt.Sprintf("Hello, %s!", string(input))), nil

}

EOF


cat > test-function/go.mod << 'EOF'

module example.com/hello


go 1.21

EOF
```

3. **Package the function**:

```bash
curl -X POST http://localhost:8080/v1/functions \

  -F "function.yaml=@test-function/function.yaml" \

  -F "code=@test-function/main.go" \

  -F "code=@test-function/go.mod"
```

Expected Response:

```json
{

  "function_id": "hello-world",

  "version": "hello-world-a1b2c3d4e5f6",

  "artifact_hash": "a1b2c3d4e5f67890...",

  "artifact_size": 2850367

}
```

4. **Verify artifact storage**:

```bash
# Check artifact exists in storage
ls -la /tmp/func-storage/artifacts/a1/b2/c3/


# List function versions
curl http://localhost:8080/v1/functions/hello-world/versions
```

5. **Signs of Correct Implementation**:
    - Artifact file exists at hash-based path
    - Binary is Linux executable (`file` shows "ELF 64-bit LSB executable")
    - Binary is stripped (no debug symbols, reduced size)
    - Registry shows the function version with correct metadata
    - Re-uploading identical code returns same hash (deduplication works)
6. **Common Issues and Fixes**:
    - **"Handler not found"**: Check handler parsing logic; ensure package path matches directory structure.
    - **"Build failed"**: Check Go installation and environment variables for cross-compilation.
    - **"Artifact too large"**: Verify `-ldflags="-s -w"` is being used; strip debug symbols.
    - **"Dependency resolution failed"**: Ensure `go.mod` exists or is created automatically for simple functions.

Component Design: Execution Environment

> **Milestone(s):** Milestone 2

Mental Model: The Pop-Up Shop

Imagine you're opening a series of temporary, identical retail shops in a busy market district. Each shop must be fully self-contained with all necessary inventory, equipment, and staff. When a customer arrives, you need to instantly open a shop, serve them in complete privacy, then close it down without leaving any trace. This is exactly what our execution environment does for serverless functions.

Think of our system as a **pop-up shop management company**:

- **Shop Blueprint** = The base container image with language runtime and system dependencies
- **Shop Inventory** = The function code and its specific dependencies
- **Shop Staff** = The handler process that actually serves requests
- **Shop Security** = Resource limits (budget), time limits (operating hours), and isolation walls (private spaces)
- **Shop Manager** = Our ContainerManager interface that handles setup, operations, and teardown

When a request arrives (a customer), we either reuse an already-open shop (warm instance) or rapidly set up a new one from the blueprint (cold start). Each shop operates independently—customers in one shop can't see or interfere with customers in another, even if they're right next door. After serving its customer, the shop stays open for a short time in case more customers arrive, then closes completely, returning all resources to the market.

This mental model helps us understand the **ephemeral**, **isolated**, and **resource-bound** nature of serverless execution environments. We're not running permanent servers but creating temporary, fully-contained execution spaces that appear and disappear on demand.

Container Manager Interface

The `ContainerManager` interface abstracts the underlying isolation technology (containers, microVMs, etc.) and provides a uniform API for managing function execution environments. This interface sits at the heart of our execution plane, responsible for creating, managing, and destroying isolated environments with precise resource controls.

Method	Parameters	Returns	Description
`CreateContainer`	`ctx context.Context`, `req CreateContainerRequest`	`(*Container, error)`	Creates a new isolated environment with the specified configuration but doesn't start it
`StartContainer`	`ctx context.Context`, `containerID string`	`error`	Begins execution of a previously created container
`ExecuteInContainer`	`ctx context.Context`, `containerID string`, `cmd []string`, `input io.Reader`, `output io.Writer`	`(int, error)`	Runs a command inside a running container, capturing stdin/stdout and returning exit code
`StopContainer`	`ctx context.Context`, `containerID string`, `timeout time.Duration`	`error`	Gracefully stops a running container with SIGTERM then SIGKILL
`RemoveContainer`	`ctx context.Context`, `containerID string`	`error`	Completely cleans up all container resources (network, storage, process)
`GetContainerStats`	`ctx context.Context`, `containerID string`	`(*ContainerStats, error)`	Returns current resource usage statistics for monitoring and scaling
`ListContainers`	`ctx context.Context`, `filter ListFilter`	`([]Container, error)`	Returns all managed containers matching filter criteria
`HealthCheck`	`ctx context.Context`	`error`	Verifies the underlying container runtime is operational

The `CreateContainerRequest` structure defines the complete specification for an isolated environment:

Field	Type	Description
`Image`	`string`	Base container image (e.g., "golang:1.21-alpine")
`Cmd`	`[]string`	Command to execute when container starts
`Env`	`[]string`	Environment variables in "KEY=VALUE" format
`Labels`	`map[string]string`	Metadata labels for tracking and filtering
`CPUQuota`	`int64`	CPU allocation in microseconds per second (e.g., 500000 = 0.5 CPU)
`MemoryLimit`	`int64`	Maximum memory in bytes
`NetworkMode`	`string`	Network isolation mode ("none", "bridge", "host")
`Binds`	`[]string`	Volume mounts in "host:container:options" format
`WorkingDir`	`string`	Working directory inside container
`User`	`string`	User:group to run as (e.g., "1000:1000")
`SecurityOpts`	`[]string`	Security options (seccomp, AppArmor, capabilities)
`ExposedPorts`	`map[string]struct{}`	Ports to expose internally
`PortBindings`	`map[string][]PortBinding`	Port mappings to host

When a function needs to execute, the system follows this workflow:

1. **Environment Creation**: The `ContainerManager` receives a `CreateContainerRequest` with the function's base image, resource limits, and configuration.
2. **Resource Allocation**: The underlying runtime (e.g., Docker) creates a container with the specified cgroups, namespaces, and security profiles.
3. **Code Injection**: The function artifact (from Milestone 1) is mounted into the container's filesystem.
4. **Process Start**: The container starts, initializing the language runtime and loading dependencies.
5. **Execution**: The handler process begins listening for invocations via HTTP or stdin.
6. **Monitoring**: Resource usage is continuously tracked via `GetContainerStats`.
7. **Cleanup**: After the function completes or times out, the container is stopped and removed.

ADR: Container vs. MicroVM Isolation

> **Decision: Use Linux Containers with Enhanced Security Hardening**
>
> **Context**: We need strong **isolation** between untrusted user functions while maintaining **fast startup times** for cold starts. The isolation boundary must prevent functions from accessing host resources, other functions' memory/files, and must enforce strict resource limits (CPU, memory, network). We evaluated two primary approaches: traditional Linux containers (namespaces + cgroups) and lightweight virtual machines (microVMs).
>
> **Options Considered**:
>
> 1. **Linux Containers with Enhanced Hardening**: Use Docker/containerd with namespace isolation, cgroups for resources, plus seccomp, AppArmor, and capability dropping
> 2. **Firecracker MicroVMs**: Use lightweight KVM-based virtual machines with minimal device emulation and optimized boot time
> 3. **gVisor/Sandboxed Containers**: Use a userspace kernel that intercepts system calls for stronger isolation than containers but weaker than VMs
>
> **Decision**: We chose **Option 1: Linux Containers with Enhanced Hardening** as our primary isolation mechanism.
>
> **Rationale**:
>
> 1. **Cold Start Performance**: Containers start in 100-500ms vs. microVMs at 100-300ms with snapshotting. Without snapshotting, microVMs take 1-3 seconds. Our goal of sub-100ms cold starts (Milestone 3) is more achievable with container checkpoint/restore (CRIU).
> 2. **Resource Efficiency**: Containers share the host kernel, allowing for higher density (more functions per host) and lower memory overhead (~5MB per container vs. ~5MB + VM memory overhead).
> 3. **Maturity and Tooling**: The container ecosystem (Docker, containerd, Kubernetes) provides battle-tested tooling for lifecycle management, networking, and storage.
> 4. **Sufficient Isolation for Most Workloads**: With proper security hardening (no root, read-only rootfs, dropped capabilities, seccomp, AppArmor), containers provide adequate isolation for multi-tenant serverless functions where users don't control the underlying infrastructure.
> 5. **Simpler Implementation**: Container APIs are standardized (OCI) and Go libraries (Docker SDK) are mature, reducing implementation complexity.
>
> **Consequences**:
>
> - **Positive**: Faster cold starts, higher density, simpler implementation, extensive tooling
> - **Negative**: Requires careful security configuration to prevent kernel-level escapes
> - **Mitigation**: We implement defense-in-depth with multiple security layers and will support microVMs as an optional, more secure backend for high-risk workloads

Isolation Technology	Startup Time (cold)	Isolation Strength	Memory Overhead	Implementation Complexity	Best For
Linux Containers (hardened)	100-500ms	Moderate (kernel sharing)	~5-50MB	Medium	General-purpose serverless, fast cold starts
Firecracker MicroVMs	100-300ms (with snapshot) 1-3s (fresh)	Strong (hardware virtualization)	~5MB + VM memory	High	Security-critical multi-tenant, financial/healthcare workloads
gVisor/Sandboxed	200-800ms	Strong (userspace kernel)	~20-100MB	Medium-High	Additional security without full virtualization

We'll design the `ContainerManager` interface to be pluggable, allowing us to implement a `FirecrackerManager` in the future while maintaining the same API. The initial implementation will use Docker via the `DockerManager` struct.

Common Pitfalls in Execution Environment

⚠️ Pitfall: Misconfigured Resource Limits Causing Premature Termination

Description: Setting memory limits too close to the function's actual usage, or not accounting for runtime overhead (JVM heap, Python interpreter, etc.), causes functions to be killed by the OOM (Out-Of-Memory) killer before completing their work.

Why It's Wrong: Functions fail unpredictably with cryptic "signal: killed" errors instead of clean memory exhaustion errors. This makes debugging difficult and reduces platform reliability.

How to Fix:

1. Always add a memory buffer (e.g., 20-30%) above the function's declared memory limit for runtime overhead
2. Use memory-swap limits to disable swapping (set `MemorySwap` = `MemoryLimit`) to ensure predictable performance
3. Monitor actual usage with `GetContainerStats` and adjust default limits based on runtime
4. Provide clear error messages when containers are OOM-killed

⚠️ Pitfall: Zombie Containers Accumulating and Exhausting Host Resources

Description: When containers aren't properly cleaned up after function execution (due to crashes, timeouts, or programming errors), they accumulate as "zombie" processes consuming memory, file descriptors, and other system resources.

Why It's Wrong: Resource leaks eventually cause host exhaustion, leading to "no space left on device" or "too many open files" errors that affect all functions on the host.

How to Fix:

1. Implement a garbage collection routine in the `ContainerManager` that periodically lists all containers and removes those not in the active inventory
2. Use container labels to track ownership and expected lifetime
3. Always call `RemoveContainer` in a `defer` statement or finally block after container use
4. Set container auto-remove flags where supported by the runtime

⚠️ Pitfall: Inadequate Sandboxing Allowing Privilege Escalation

Description: Running containers as root, with excessive Linux capabilities, or without seccomp/AppArmor profiles allows malicious functions to escape isolation and access the host or other functions.

Why It's Wrong: Complete security breach violating multi-tenancy. Functions can read secrets, modify other functions' code, or attack the host system.

How to Fix:

1. Always run containers as non-root users (specify `User: "1000:1000"` in `CreateContainerRequest`)
2. Drop all capabilities and add back only specifically required ones: `SecurityOpts: ["no-new-privileges", "cap-drop=ALL"]`
3. Apply restrictive seccomp profiles that block dangerous syscalls (clone, ptrace, kernel module operations)
4. Use read-only root filesystem where possible, with only `/tmp` writable
5. Disable inter-container communication at the network level

⚠️ Pitfall: Shared Mutable State Between Invocations Causing Data Leaks

Description: Reusing the same container for multiple invocations without resetting the filesystem or memory state allows one invocation to see another's data.

Why It's Wrong: Violates the serverless principle that functions should be stateless. User A's data could be visible to User B if they happen to use the same container.

How to Fix:

1. Treat containers as single-use for security-critical workloads, creating a new container per invocation
2. For performance (warm starts), only reuse containers for the same function version and reset writable areas between invocations
3. Use overlay filesystems with a fresh upper layer for each invocation
4. Clear environment variables, process memory is harder—consider it contaminated and don't reuse for different users

Implementation Guidance for Execution Environment

Technology Recommendations

Component	Simple Option	Advanced Option
Container Runtime	Docker Engine (docker.io)	containerd with CRI plugin
Container SDK	Docker Go SDK (github.com/docker/docker/client)	Direct containerd Go client (github.com/containerd/containerd)
Resource Limits	cgroups v2 via container runtime	Manual cgroup setup via `github.com/containerd/cgroups`
Security Profiles	Default Docker seccomp + custom AppArmor	Custom seccomp JSON profiles + SELinux

Recommended File/Module Structure

```
project-root/
├── cmd/
│   └── server/
│       └── main.go                # Entry point
├── internal/
│   ├── containermanager/          # Container isolation subsystem
│   │   ├── manager.go             # ContainerManager interface
│   │   ├── docker_manager.go      # DockerManager implementation
│   │   ├── firecracker_manager.go  # Future: Firecracker implementation
│   │   ├── cgroup_helper.go       # Low-level cgroup utilities
│   │   ├── security/              # Security profiles
│   │   │   ├── seccomp.go         # Seccomp profile generator
│   │   │   └── apparmor.go        # AppArmor profile templates
│   │   └── tests/
│   │       └── manager_test.go    # Integration tests
│   ├── execution/                 # Execution coordination
│   │   ├── executor.go            # Orchestrates container lifecycle per function
│   │   └── runtime/               # Language-specific runtime configurations
│   │       ├── golang.go          # Go runtime setup
│   │       ├── python.go          # Python runtime setup
│   │       └── nodejs.go          # Node.js runtime setup
│   └── metrics/                   # Metrics collection
│       └── container_metrics.go   # Container-specific metrics
└── pkg/
    └── types/                     # Shared types (already defined)
        └── container.go           # Container, CreateContainerRequest types
```

Infrastructure Starter Code

cgroup_helper.go - Low-level cgroup utilities for manual resource limiting:

```go
package containermanager

import (
	"fmt"
	"os"
	"path/filepath"
	"strconv"

	"github.com/containerd/cgroups/v3"
	"github.com/containerd/cgroups/v3/cgroup1"
	cgroupv2 "github.com/containerd/cgroups/v3/cgroup2"
)

// CGroupHelper provides utilities for managing cgroups manually
// Used when container runtime doesn't provide sufficient control
type CGroupHelper struct {
	cgroupPath string
	isV2       bool
}

// NewCGroupHelper creates a cgroup at the specified path
func NewCGroupHelper(path string) (*CGroupHelper, error) {
	// Detect cgroup version
	isV2 := cgroups.Mode() == cgroups.Unified

	return &CGroupHelper{
		cgroupPath: path,
		isV2:       isV2,
	}, nil
}

// SetMemoryLimit configures memory limit and swap (if supported)
func (h *CGroupHelper) SetMemoryLimit(limitBytes int64, swapBytes int64) error {
	if h.isV2 {
		return h.setMemoryLimitV2(limitBytes)
	}
	return h.setMemoryLimitV1(limitBytes, swapBytes)
}

// setMemoryLimitV1 sets memory limit for cgroup v1
func (h *CGroupHelper) setMemoryLimitV1(limitBytes int64, swapBytes int64) error {
	memoryPath := filepath.Join("/sys/fs/cgroup/memory", h.cgroupPath)
```

```
        // Create cgroup directory if it doesn't exist
        if err := os.MkdirAll(memoryPath, 0755); err != nil {
                return fmt.Errorf("failed to create cgroup dir: %w", err)
        }

        // Set memory limit
        limitFile := filepath.Join(memoryPath, "memory.limit_in_bytes")
        if err := os.WriteFile(limitFile, []byte(strconv.FormatInt(limitBytes, 10)), 0644); err != nil {
                return fmt.Errorf("failed to set memory limit: %w", err)
        }

        // Set memory+swap limit (must be >= memory limit)
        if swapBytes > 0 {
                memswFile := filepath.Join(memoryPath, "memory.memsw.limit_in_bytes")
                if err := os.WriteFile(memswFile, []byte(strconv.FormatInt(swapBytes, 10)), 0644); err != nil {
                        // Some systems don't have swap accounting enabled
                        if !os.IsNotExist(err) {
                                return fmt.Errorf("failed to set swap limit: %w", err)
                        }
                }
        }

        return nil
}

// setMemoryLimitV2 sets memory limit for cgroup v2
func (h *CGroupHelper) setMemoryLimitV2(limitBytes int64) error {
        // For cgroup v2, we use the containerd library
        res := cgroupv2.Resources{
                Memory: &cgroupv2.Memory{
                        Max: cgroupv2.NewPointer(uint64(limitBytes)),
                },
        }

        cg, err := cgroupv2.NewManager("/sys/fs/cgroup", h.cgroupPath, &res)
        if err != nil {
                return fmt.Errorf("failed to create cgroup v2: %w", err)
        }
        defer cg.Delete()

        return nil
```

```
}

// AddProcess adds a process to the cgroup
func (h *CGroupHelper) AddProcess(pid int) error {
        if h.isV2 {
                return h.addProcessV2(pid)
        }
        return h.addProcessV1(pid)
}

// addProcessV1 adds process to cgroup v1
func (h *CGroupHelper) addProcessV1(pid int) error {
        procsFile := filepath.Join("/sys/fs/cgroup/memory", h.cgroupPath, "cgroup.procs")
        return os.WriteFile(procsFile, []byte(strconv.Itoa(pid)), 0644)
}

// addProcessV2 adds process to cgroup v2
func (h *CGroupHelper) addProcessV2(pid int) error {
        // Implementation with containerd library
        res := cgroupv2.Resources{}
        cg, err := cgroupv2.NewManager("/sys/fs/cgroup", h.cgroupPath, &res)
        if err != nil {
                return err
        }
        defer cg.Delete()

        return cg.AddProc(uint64(pid))
}

// Cleanup removes the cgroup
func (h *CGroupHelper) Cleanup() error {
        if h.isV2 {
                return h.cleanupV2()
        }
        return h.cleanupV1()
}

// cleanupV1 removes cgroup v1 directory
func (h *CGroupHelper) cleanupV1() error {
        memoryPath := filepath.Join("/sys/fs/cgroup/memory", h.cgroupPath)

        // Move any remaining processes to parent cgroup
        procsFile := filepath.Join(memoryPath, "cgroup.procs")
```

```
        if procs, err := os.ReadFile(procsFile); err == nil && len(procs) > 0 {
                parentProcs := filepath.Join(filepath.Dir(memoryPath), "cgroup.procs")
                os.WriteFile(parentProcs, procs, 0644)
        }

        return os.RemoveAll(memoryPath)
}

// cleanupV2 removes cgroup v2
func (h *CGroupHelper) cleanupV2() error {
        cg, err := cgroupv2.Load("/sys/fs/cgroup", h.cgroupPath)
        if err != nil {
                // Already cleaned up
                return nil
        }
        return cg.Delete()
}
```

security/seccomp.go - Default seccomp profile for serverless functions:

```go
package security

import (
	"encoding/json"
	"fmt"
)

// SeccompProfile defines a restrictive seccomp filter for serverless functions
type SeccompProfile struct {
	DefaultAction string          `json:"defaultAction"`
	Architectures []string        `json:"architectures"`
	Syscalls      []SeccompSyscall `json:"syscalls"`
}

// SeccompSyscall defines a single syscall rule
type SeccompSyscall struct {
	Name   string   `json:"name"`
	Action string   `json:"action"`
	Args   []SeccompArg `json:"args,omitempty"`
}

// SeccompArg defines an argument filter for syscalls
type SeccompArg struct {
	Index    uint     `json:"index"`
	Value    uint64   `json:"value"`
	ValueTwo uint64   `json:"valueTwo,omitempty"`
	Op       string   `json:"op"`
}

// DefaultSeccompProfile returns a restrictive seccomp profile for serverless functions
func DefaultSeccompProfile() (string, error) {
	profile := SeccompProfile{
		DefaultAction: "SCMP_ACT_ERRNO",
		Architectures: []string{"SCMP_ARCH_X86_64", "SCMP_ARCH_X86", "SCMP_ARCH_X32"},
		Syscalls: []SeccompSyscall{
			// Basic required syscalls
			{Name: "accept", Action: "SCMP_ACT_ALLOW"},
			{Name: "accept4", Action: "SCMP_ACT_ALLOW"},
			{Name: "access", Action: "SCMP_ACT_ALLOW"},
			{Name: "alarm", Action: "SCMP_ACT_ALLOW"},
			{Name: "bind", Action: "SCMP_ACT_ALLOW"},
			{Name: "brk", Action: "SCMP_ACT_ALLOW"},
			{Name: "capget", Action: "SCMP_ACT_ALLOW"},
```

```
{Name: "capset", Action: "SCMP_ACT_ALLOW"},

{Name: "chdir", Action: "SCMP_ACT_ALLOW"},

{Name: "chmod", Action: "SCMP_ACT_ALLOW"},

{Name: "chown", Action: "SCMP_ACT_ALLOW"},

{Name: "chown32", Action: "SCMP_ACT_ALLOW"},

{Name: "clock_getres", Action: "SCMP_ACT_ALLOW"},

{Name: "clock_gettime", Action: "SCMP_ACT_ALLOW"},

{Name: "clock_nanosleep", Action: "SCMP_ACT_ALLOW"},

{Name: "clone", Action: "SCMP_ACT_ALLOW", Args: []SeccompArg{

        {Index: 0, Value: 2114060288, Op: "SCMP_CMP_MASKED_EQ"},

}},

{Name: "close", Action: "SCMP_ACT_ALLOW"},

{Name: "connect", Action: "SCMP_ACT_ALLOW"},

{Name: "copy_file_range", Action: "SCMP_ACT_ALLOW"},

{Name: "creat", Action: "SCMP_ACT_ALLOW"},

{Name: "dup", Action: "SCMP_ACT_ALLOW"},

{Name: "dup2", Action: "SCMP_ACT_ALLOW"},

{Name: "dup3", Action: "SCMP_ACT_ALLOW"},

{Name: "epoll_create", Action: "SCMP_ACT_ALLOW"},

{Name: "epoll_create1", Action: "SCMP_ACT_ALLOW"},

{Name: "epoll_ctl", Action: "SCMP_ACT_ALLOW"},

{Name: "epoll_pwait", Action: "SCMP_ACT_ALLOW"},

{Name: "epoll_wait", Action: "SCMP_ACT_ALLOW"},

{Name: "eventfd", Action: "SCMP_ACT_ALLOW"},

{Name: "eventfd2", Action: "SCMP_ACT_ALLOW"},

{Name: "execve", Action: "SCMP_ACT_ALLOW"},

{Name: "execveat", Action: "SCMP_ACT_ALLOW"},

{Name: "exit", Action: "SCMP_ACT_ALLOW"},

{Name: "exit_group", Action: "SCMP_ACT_ALLOW"},

{Name: "faccessat", Action: "SCMP_ACT_ALLOW"},

{Name: "fadvise64", Action: "SCMP_ACT_ALLOW"},

{Name: "fallocate", Action: "SCMP_ACT_ALLOW"},

{Name: "fchdir", Action: "SCMP_ACT_ALLOW"},

{Name: "fchmod", Action: "SCMP_ACT_ALLOW"},

{Name: "fchmodat", Action: "SCMP_ACT_ALLOW"},

{Name: "fchown", Action: "SCMP_ACT_ALLOW"},

{Name: "fchown32", Action: "SCMP_ACT_ALLOW"},

{Name: "fchownat", Action: "SCMP_ACT_ALLOW"},

{Name: "fcntl", Action: "SCMP_ACT_ALLOW"},

{Name: "fcntl64", Action: "SCMP_ACT_ALLOW"},
```

```
                {Name: "fdatasync", Action: "SCMP_ACT_ALLOW"},
                {Name: "fgetxattr", Action: "SCMP_ACT_ALLOW"},
                {Name: "flistxattr", Action: "SCMP_ACT_ALLOW"},
                {Name: "flock", Action: "SCMP_ACT_ALLOW"},
                {Name: "fork", Action: "SCMP_ACT_ALLOW"},
                {Name: "fremovexattr", Action: "SCMP_ACT_ALLOW"},
                {Name: "fsetxattr", Action: "SCMP_ACT_ALLOW"},
                {Name: "fstat", Action: "SCMP_ACT_ALLOW"},
                {Name: "fstat64", Action: "SCMP_ACT_ALLOW"},
                {Name: "fstatat64", Action: "SCMP_ACT_ALLOW"},
                {Name: "fstatfs", Action: "SCMP_ACT_ALLOW"},
                {Name: "fstatfs64", Action: "SCMP_ACT_ALLOW"},
                {Name: "fsync", Action: "SCMP_ACT_ALLOW"},
                {Name: "ftruncate", Action: "SCMP_ACT_ALLOW"},
                {Name: "ftruncate64", Action: "SCMP_ACT_ALLOW"},
                {Name: "futex", Action: "SCMP_ACT_ALLOW"},
                {Name: "getcwd", Action: "SCMP_ACT_ALLOW"},
                {Name: "getdents", Action: "SCMP_ACT_ALLOW"},
                {Name: "getdents64", Action: "SCMP_ACT_ALLOW"},
                {Name: "getegid", Action: "SCMP_ACT_ALLOW"},
                {Name: "getegid32", Action: "SCMP_ACT_ALLOW"},
                {Name: "geteuid", Action: "SCMP_ACT_ALLOW"},
                {Name: "geteuid32", Action: "SCMP_ACT_ALLOW"},
                {Name: "getgid", Action: "SCMP_ACT_ALLOW"},
                {Name: "getgid32", Action: "SCMP_ACT_ALLOW"},
                {Name: "getgroups", Action: "SCMP_ACT_ALLOW"},
                {Name: "getgroups32", Action: "SCMP_ACT_ALLOW"},
                {Name: "getitimer", Action: "SCMP_ACT_ALLOW"},
                {Name: "getpeername", Action: "SCMP_ACT_ALLOW"},
                {Name: "getpgid", Action: "SCMP_ACT_ALLOW"},
                {Name: "getpgrp", Action: "SCMP_ACT_ALLOW"},
                {Name: "getpid", Action: "SCMP_ACT_ALLOW"},
                {Name: "getppid", Action: "SCMP_ACT_ALLOW"},
                {Name: "getpriority", Action: "SCMP_ACT_ALLOW"},
                {Name: "getrandom", Action: "SCMP_ACT_ALLOW"},
                {Name: "getresgid", Action: "SCMP_ACT_ALLOW"},
                {Name: "getresgid32", Action: "SCMP_ACT_ALLOW"},
                {Name: "getresuid", Action: "SCMP_ACT_ALLOW"},
                {Name: "getresuid32", Action: "SCMP_ACT_ALLOW"},
                {Name: "getrlimit", Action: "SCMP_ACT_ALLOW"},
```

```
                {Name: "get_robust_list", Action: "SCMP_ACT_ALLOW"},
                {Name: "getrusage", Action: "SCMP_ACT_ALLOW"},
                {Name: "getsid", Action: "SCMP_ACT_ALLOW"},
                {Name: "getsockname", Action: "SCMP_ACT_ALLOW"},
                {Name: "getsockopt", Action: "SCMP_ACT_ALLOW"},
                {Name: "gettid", Action: "SCMP_ACT_ALLOW"},
                {Name: "gettimeofday", Action: "SCMP_ACT_ALLOW"},
                {Name: "getuid", Action: "SCMP_ACT_ALLOW"},
                {Name: "getuid32", Action: "SCMP_ACT_ALLOW"},
                {Name: "getxattr", Action: "SCMP_ACT_ALLOW"},
                {Name: "inotify_add_watch", Action: "SCMP_ACT_ALLOW"},
                {Name: "inotify_init", Action: "SCMP_ACT_ALLOW"},
                {Name: "inotify_init1", Action: "SCMP_ACT_ALLOW"},
                {Name: "inotify_rm_watch", Action: "SCMP_ACT_ALLOW"},
                {Name: "io_cancel", Action: "SCMP_ACT_ALLOW"},
                {Name: "ioctl", Action: "SCMP_ACT_ALLOW"},
                {Name: "io_destroy", Action: "SCMP_ACT_ALLOW"},
                {Name: "io_getevents", Action: "SCMP_ACT_ALLOW"},
                {Name: "ioprio_get", Action: "SCMP_ACT_ALLOW"},
                {Name: "ioprio_set", Action: "SCMP_ACT_ALLOW"},
                {Name: "io_setup", Action: "SCMP_ACT_ALLOW"},
                {Name: "io_submit", Action: "SCMP_ACT_ALLOW"},
                {Name: "ipc", Action: "SCMP_ACT_ALLOW"},
                {Name: "kill", Action: "SCMP_ACT_ALLOW"},
                {Name: "lchown", Action: "SCMP_ACT_ALLOW"},
                {Name: "lchown32", Action: "SCMP_ACT_ALLOW"},
                {Name: "lgetxattr", Action: "SCMP_ACT_ALLOW"},
                {Name: "link", Action: "SCMP_ACT_ALLOW"},
                {Name: "linkat", Action: "SCMP_ACT_ALLOW"},
                {Name: "listen", Action: "SCMP_ACT_ALLOW"},
                {Name: "listxattr", Action: "SCMP_ACT_ALLOW"},
                {Name: "llistxattr", Action: "SCMP_ACT_ALLOW"},
                {Name: "_llseek", Action: "SCMP_ACT_ALLOW"},
                {Name: "lremovexattr", Action: "SCMP_ACT_ALLOW"},
                {Name: "lseek", Action: "SCMP_ACT_ALLOW"},
                {Name: "lsetxattr", Action: "SCMP_ACT_ALLOW"},
                {Name: "lstat", Action: "SCMP_ACT_ALLOW"},
                {Name: "lstat64", Action: "SCMP_ACT_ALLOW"},
                {Name: "madvise", Action: "SCMP_ACT_ALLOW"},
                {Name: "memfd_create", Action: "SCMP_ACT_ALLOW"},
```

```
                {Name: "mincore", Action: "SCMP_ACT_ALLOW"},
                {Name: "mkdir", Action: "SCMP_ACT_ALLOW"},
                {Name: "mkdirat", Action: "SCMP_ACT_ALLOW"},
                {Name: "mknod", Action: "SCMP_ACT_ALLOW"},
                {Name: "mknodat", Action: "SCMP_ACT_ALLOW"},
                {Name: "mlock", Action: "SCMP_ACT_ALLOW"},
                {Name: "mlock2", Action: "SCMP_ACT_ALLOW"},
                {Name: "mlockall", Action: "SCMP_ACT_ALLOW"},
                {Name: "mmap", Action: "SCMP_ACT_ALLOW"},
                {Name: "mmap2", Action: "SCMP_ACT_ALLOW"},
                {Name: "mprotect", Action: "SCMP_ACT_ALLOW"},
                {Name: "mq_getsetattr", Action: "SCMP_ACT_ALLOW"},
                {Name: "mq_notify", Action: "SCMP_ACT_ALLOW"},
                {Name: "mq_open", Action: "SCMP_ACT_ALLOW"},
                {Name: "mq_timedreceive", Action: "SCMP_ACT_ALLOW"},
                {Name: "mq_timedsend", Action: "SCMP_ACT_ALLOW"},
                {Name: "mq_unlink", Action: "SCMP_ACT_ALLOW"},
                {Name: "mremap", Action: "SCMP_ACT_ALLOW"},
                {Name: "msgctl", Action: "SCMP_ACT_ALLOW"},
                {Name: "msgget", Action: "SCMP_ACT_ALLOW"},
                {Name: "msgrcv", Action: "SCMP_ACT_ALLOW"},
                {Name: "msgsnd", Action: "SCMP_ACT_ALLOW"},
                {Name: "msync", Action: "SCMP_ACT_ALLOW"},
                {Name: "munlock", Action: "SCMP_ACT_ALLOW"},
                {Name: "munlockall", Action: "SCMP_ACT_ALLOW"},
                {Name: "munmap", Action: "SCMP_ACT_ALLOW"},
                {Name: "nanosleep", Action: "SCMP_ACT_ALLOW"},
                {Name: "newfstatat", Action: "SCMP_ACT_ALLOW"},
                {Name: "_newselect", Action: "SCMP_ACT_ALLOW"},
                {Name: "open", Action: "SCMP_ACT_ALLOW"},
                {Name: "openat", Action: "SCMP_ACT_ALLOW"},
                {Name: "pause", Action: "SCMP_ACT_ALLOW"},
                {Name: "pipe", Action: "SCMP_ACT_ALLOW"},
                {Name: "pipe2", Action: "SCMP_ACT_ALLOW"},
                {Name: "poll", Action: "SCMP_ACT_ALLOW"},
                {Name: "ppoll", Action: "SCMP_ACT_ALLOW"},
                {Name: "prctl", Action: "SCMP_ACT_ALLOW"},
                {Name: "pread64", Action: "SCMP_ACT_ALLOW"},
                {Name: "preadv", Action: "SCMP_ACT_ALLOW"},
                {Name: "preadv2", Action: "SCMP_ACT_ALLOW"},
```

```
                {Name: "prlimit64", Action: "SCMP_ACT_ALLOW"},
                {Name: "pselect6", Action: "SCMP_ACT_ALLOW"},
                {Name: "pwrite64", Action: "SCMP_ACT_ALLOW"},
                {Name: "pwritev", Action: "SCMP_ACT_ALLOW"},
                {Name: "pwritev2", Action: "SCMP_ACT_ALLOW"},
                {Name: "read", Action: "SCMP_ACT_ALLOW"},
                {Name: "readahead", Action: "SCMP_ACT_ALLOW"},
                {Name: "readlink", Action: "SCMP_ACT_ALLOW"},
                {Name: "readlinkat", Action: "SCMP_ACT_ALLOW"},
                {Name: "readv", Action: "SCMP_ACT_ALLOW"},
                {Name: "recv", Action: "SCMP_ACT_ALLOW"},
                {Name: "recvfrom", Action: "SCMP_ACT_ALLOW"},
                {Name: "recvmmsg", Action: "SCMP_ACT_ALLOW"},
                {Name: "recvmsg", Action: "SCMP_ACT_ALLOW"},
                {Name: "remap_file_pages", Action: "SCMP_ACT_ALLOW"},
                {Name: "removexattr", Action: "SCMP_ACT_ALLOW"},
                {Name: "rename", Action: "SCMP_ACT_ALLOW"},
                {Name: "renameat", Action: "SCMP_ACT_ALLOW"},
                {Name: "renameat2", Action: "SCMP_ACT_ALLOW"},
                {Name: "rmdir", Action: "SCMP_ACT_ALLOW"},
                {Name: "rt_sigaction", Action: "SCMP_ACT_ALLOW"},
                {Name: "rt_sigpending", Action: "SCMP_ACT_ALLOW"},
                {Name: "rt_sigprocmask", Action: "SCMP_ACT_ALLOW"},
                {Name: "rt_sigqueueinfo", Action: "SCMP_ACT_ALLOW"},
                {Name: "rt_sigreturn", Action: "SCMP_ACT_ALLOW"},
                {Name: "rt_sigsuspend", Action: "SCMP_ACT_ALLOW"},
                {Name: "rt_sigtimedwait", Action: "SCMP_ACT_ALLOW"},
                {Name: "rt_tgsigqueueinfo", Action: "SCMP_ACT_ALLOW"},
                {Name: "sched_getaffinity", Action: "SCMP_ACT_ALLOW"},
                {Name: "sched_getparam", Action: "SCMP_ACT_ALLOW"},
                {Name: "sched_getscheduler", Action: "SCMP_ACT_ALLOW"},
                {Name: "sched_get_priority_max", Action: "SCMP_ACT_ALLOW"},
                {Name: "sched_get_priority_min", Action: "SCMP_ACT_ALLOW"},
                {Name: "sched_rr_get_interval", Action: "SCMP_ACT_ALLOW"},
                {Name: "sched_setaffinity", Action: "SCMP_ACT_ALLOW"},
                {Name: "sched_setparam", Action: "SCMP_ACT_ALLOW"},
                {Name: "sched_setscheduler", Action: "SCMP_ACT_ALLOW"},
                {Name: "sched_yield", Action: "SCMP_ACT_ALLOW"},
                {Name: "seccomp", Action: "SCMP_ACT_ALLOW"},
                {Name: "select", Action: "SCMP_ACT_ALLOW"},
```

```
                {Name: "semctl", Action: "SCMP_ACT_ALLOW"},
                {Name: "semget", Action: "SCMP_ACT_ALLOW"},
                {Name: "semop", Action: "SCMP_ACT_ALLOW"},
                {Name: "semtimedop", Action: "SCMP_ACT_ALLOW"},
                {Name: "send", Action: "SCMP_ACT_ALLOW"},
                {Name: "sendfile", Action: "SCMP_ACT_ALLOW"},
                {Name: "sendfile64", Action: "SCMP_ACT_ALLOW"},
                {Name: "sendmmsg", Action: "SCMP_ACT_ALLOW"},
                {Name: "sendmsg", Action: "SCMP_ACT_ALLOW"},
                {Name: "sendto", Action: "SCMP_ACT_ALLOW"},
                {Name: "setfsgid", Action: "SCMP_ACT_ALLOW"},
                {Name: "setfsgid32", Action: "SCMP_ACT_ALLOW"},
                {Name: "setfsuid", Action: "SCMP_ACT_ALLOW"},
                {Name: "setfsuid32", Action: "SCMP_ACT_ALLOW"},
                {Name: "setgid", Action: "SCMP_ACT_ALLOW"},
                {Name: "setgid32", Action: "SCMP_ACT_ALLOW"},
                {Name: "setgroups", Action: "SCMP_ACT_ALLOW"},
                {Name: "setgroups32", Action: "SCMP_ACT_ALLOW"},
                {Name: "setitimer", Action: "SCMP_ACT_ALLOW"},
                {Name: "setpgid", Action: "SCMP_ACT_ALLOW"},
                {Name: "setpriority", Action: "SCMP_ACT_ALLOW"},
                {Name: "setregid", Action: "SCMP_ACT_ALLOW"},
                {Name: "setregid32", Action: "SCMP_ACT_ALLOW"},
                {Name: "setresgid", Action: "SCMP_ACT_ALLOW"},
                {Name: "setresgid32", Action: "SCMP_ACT_ALLOW"},
                {Name: "setresuid", Action: "SCMP_ACT_ALLOW"},
                {Name: "setresuid32", Action: "SCMP_ACT_ALLOW"},
                {Name: "setreuid", Action: "SCMP_ACT_ALLOW"},
                {Name: "setreuid32", Action: "SCMP_ACT_ALLOW"},
                {Name: "setrlimit", Action: "SCMP_ACT_ALLOW"},
                {Name: "setsid", Action: "SCMP_ACT_ALLOW"},
                {Name: "setsockopt", Action: "SCMP_ACT_ALLOW"},
                {Name: "set_tid_address", Action: "SCMP_ACT_ALLOW"},
                {Name: "setuid", Action: "SCMP_ACT_ALLOW"},
                {Name: "setuid32", Action: "SCMP_ACT_ALLOW"},
                {Name: "setxattr", Action: "SCMP_ACT_ALLOW"},
                {Name: "shmat", Action: "SCMP_ACT_ALLOW"},
                {Name: "shmctl", Action: "SCMP_ACT_ALLOW"},
                {Name: "shmdt", Action: "SCMP_ACT_ALLOW"},
                {Name: "shmget", Action: "SCMP_ACT_ALLOW"},
```

```
                {Name: "shutdown", Action: "SCMP_ACT_ALLOW"},
                {Name: "sigaltstack", Action: "SCMP_ACT_ALLOW"},
                {Name: "signalfd", Action: "SCMP_ACT_ALLOW"},
                {Name: "signalfd4", Action: "SCMP_ACT_ALLOW"},
                {Name: "sigreturn", Action: "SCMP_ACT_ALLOW"},
                {Name: "socket", Action: "SCMP_ACT_ALLOW"},
                {Name: "socketcall", Action: "SCMP_ACT_ALLOW"},
                {Name: "socketpair", Action: "SCMP_ACT_ALLOW"},
                {Name: "splice", Action: "SCMP_ACT_ALLOW"},
                {Name: "stat", Action: "SCMP_ACT_ALLOW"},
                {Name: "stat64", Action: "SCMP_ACT_ALLOW"},
                {Name: "statfs", Action: "SCMP_ACT_ALLOW"},
                {Name: "statfs64", Action: "SCMP_ACT_ALLOW"},
                {Name: "symlink", Action: "SCMP_ACT_ALLOW"},
                {Name: "symlinkat", Action: "SCMP_ACT_ALLOW"},
                {Name: "sync", Action: "SCMP_ACT_ALLOW"},
                {Name: "sync_file_range", Action: "SCMP_ACT_ALLOW"},
                {Name: "sysinfo", Action: "SCMP_ACT_ALLOW"},
                {Name: "tee", Action: "SCMP_ACT_ALLOW"},
                {Name: "tgkill", Action: "SCMP_ACT_ALLOW"},
                {Name: "time", Action: "SCMP_ACT_ALLOW"},
                {Name: "timer_create", Action: "SCMP_ACT_ALLOW"},
                {Name: "timer_delete", Action: "SCMP_ACT_ALLOW"},
                {Name: "timer_getoverrun", Action: "SCMP_ACT_ALLOW"},
                {Name: "timer_gettime", Action: "SCMP_ACT_ALLOW"},
                {Name: "timer_settime", Action: "SCMP_ACT_ALLOW"},
                {Name: "timerfd_create", Action: "SCMP_ACT_ALLOW"},
                {Name: "timerfd_gettime", Action: "SCMP_ACT_ALLOW"},
                {Name: "timerfd_settime", Action: "SCMP_ACT_ALLOW"},
                {Name: "times", Action: "SCMP_ACT_ALLOW"},
                {Name: "tkill", Action: "SCMP_ACT_ALLOW"},
                {Name: "truncate", Action: "SCMP_ACT_ALLOW"},
                {Name: "truncate64", Action: "SCMP_ACT_ALLOW"},
                {Name: "ugetrlimit", Action: "SCMP_ACT_ALLOW"},
                {Name: "umask", Action: "SCMP_ACT_ALLOW"},
                {Name: "uname", Action: "SCMP_ACT_ALLOW"},
                {Name: "unlink", Action: "SCMP_ACT_ALLOW"},
                {Name: "unlinkat", Action: "SCMP_ACT_ALLOW"},
                {Name: "utime", Action: "SCMP_ACT_ALLOW"},
                {Name: "utimensat", Action: "SCMP_ACT_ALLOW"},
```

```
                        {Name: "utimes", Action: "SCMP_ACT_ALLOW"},
                        {Name: "vfork", Action: "SCMP_ACT_ALLOW"},
                        {Name: "vmsplice", Action: "SCMP_ACT_ALLOW"},
                        {Name: "wait4", Action: "SCMP_ACT_ALLOW"},
                        {Name: "waitid", Action: "SCMP_ACT_ALLOW"},
                        {Name: "waitpid", Action: "SCMP_ACT_ALLOW"},
                        {Name: "write", Action: "SCMP_ACT_ALLOW"},
                        {Name: "writev", Action: "SCMP_ACT_ALLOW"},
                },
        }

        jsonData, err := json.MarshalIndent(profile, "", "  ")
        if err != nil {
                return "", fmt.Errorf("failed to marshal seccomp profile: %w", err)
        }

        return string(jsonData), nil
}
```

Core Logic Skeleton Code

docker_manager.go - Docker implementation of ContainerManager:

GO

```go
package containermanager

import (
	"context"
	"io"
	"time"

	"github.com/docker/docker/api/types"
	"github.com/docker/docker/api/types/container"
	"github.com/docker/docker/api/types/filters"
	"github.com/docker/docker/api/types/network"
	"github.com/docker/docker/client"

	"yourproject/pkg/types"
)

// DockerManager implements ContainerManager using Docker Engine
type DockerManager struct {
	client *docker.Client
}

// NewDockerManager creates a new DockerManager
func NewDockerManager() (*DockerManager, error) {
	// TODO 1: Create Docker client using client.NewClientWithOpts()
	// TODO 2: Configure client with appropriate API version and timeout
	// TODO 3: Verify connection with Ping() and return error if Docker is unavailable
	// TODO 4: Return initialized DockerManager with client
	return nil, nil
}

// CreateContainer creates a new Docker container with the specified configuration
func (dm *DockerManager) CreateContainer(ctx context.Context, req types.CreateContainerRequest) (*types.Container, error) {
	// TODO 1: Convert MemoryLimit from bytes to megabytes for Docker (divide by 1024*1024)
	// TODO 2: Prepare container configuration with:
	//         - Image: req.Image
	//         - Cmd: req.Cmd
	//         - Env: req.Env
	//         - WorkingDir: req.WorkingDir
	//         - User: req.User
	//         - Labels: req.Labels
	//         - ExposedPorts: req.ExposedPorts
	// TODO 3: Prepare host configuration with:
	//         - Memory: req.MemoryLimit
```

```
        //          - MemorySwap: req.MemoryLimit (to disable swap)
        //          - NanoCPUs: req.CPUQuota * 100 (convert to nanoseconds)
        //          - Binds: req.Binds
        //          - PortBindings: req.PortBindings
        //          - SecurityOpt: append(req.SecurityOpts, "no-new-privileges", "seccomp=...")
        //          - ReadonlyRootfs: true (except for /tmp)
        // TODO 4: Call dm.client.ContainerCreate() with configs
        // TODO 5: On success, create and return types.Container with:
        //          - ID from response
        //          - Status: StatusCreated
        //          - Labels: req.Labels
        //          - CreatedAt: time.Now()
        // TODO 6: Handle errors (image pull, invalid config, etc.)
        return nil, nil
}

// StartContainer starts a previously created container
func (dm *DockerManager) StartContainer(ctx context.Context, containerID string) error {
        // TODO 1: Call dm.client.ContainerStart() with containerID
        // TODO 2: Handle specific errors (container already running, not found)
        // TODO 3: Wait briefly to ensure container is actually running
        // TODO 4: Return nil on success, error on failure
        return nil
}

// ExecuteInContainer runs a command inside a running container
func (dm *DockerManager) ExecuteInContainer(ctx context.Context, containerID string, cmd []string, input io.Reader, output io.Writer)
(int, error) {
        // TODO 1: Create exec configuration with Cmd, User, Privileged: false
        // TODO 2: Call dm.client.ContainerExecCreate() to create exec instance
        // TODO 3: Call dm.client.ContainerExecStart() with hijacked response
        // TODO 4: Stream input to exec's stdin if provided
        // TODO 5: Stream exec's stdout/stderr to output
        // TODO 6: Wait for exec to complete with ContainerExecInspect
        // TODO 7: Return exit code and any error
        return 0, nil
}

// StopContainer gracefully stops a running container
func (dm *DockerManager) StopContainer(ctx context.Context, containerID string, timeout time.Duration) error {
        // TODO 1: Convert timeout to seconds (Docker uses seconds)
        // TODO 2: Call dm.client.ContainerStop() with containerID and timeout
```

```
        // TODO 3: If force is needed after timeout, call ContainerKill()
        // TODO 4: Return error only if both stop and kill fail
        return nil
}

// RemoveContainer cleans up container resources
func (dm *DockerManager) RemoveContainer(ctx context.Context, containerID string) error {
        // TODO 1: Set remove options: RemoveVolumes: true, Force: true, RemoveLinks: false
        // TODO 2: Call dm.client.ContainerRemove() with options
        // TODO 3: Log warning but don't fail if container doesn't exist (already removed)
        // TODO 4: Return actual error only for unexpected failures
        return nil
}

// GetContainerStats returns current resource usage statistics
func (dm *DockerManager) GetContainerStats(ctx context.Context, containerID string) (*types.ContainerStats, error) {
        // TODO 1: Call dm.client.ContainerStats() with containerID and stream: false
        // TODO 2: Decode the stats response (types.StatsJSON)
        // TODO 3: Calculate CPU percentage: (cpuDelta / systemDelta) * 100
        // TODO 4: Extract memory usage, limit, and percentage
        // TODO 5: Extract network I/O statistics
        // TODO 6: Extract block I/O statistics
        // TODO 7: Return types.ContainerStats with all calculated values
        return nil, nil
}

// ListContainers returns all managed containers matching filter criteria
func (dm *DockerManager) ListContainers(ctx context.Context, filter types.ListFilter) ([]types.Container, error) {
        // TODO 1: Build Docker filters from filter.Labels
        // TODO 2: Call dm.client.ContainerList() with All: true and filters
        // TODO 3: Convert each Docker container to types.Container
        // TODO 4: Apply status filtering if filter.Status is specified
        // TODO 5: Return slice of matching containers
        return nil, nil
}

// HealthCheck verifies Docker daemon is operational
func (dm *DockerManager) HealthCheck(ctx context.Context) error {
        // TODO 1: Call dm.client.Ping() with context
        // TODO 2: Verify API version compatibility
        // TODO 3: Check disk space if possible
        // TODO 4: Return nil if healthy, error otherwise
```

```
        return nil
}
```

executor.go - Orchestrates container lifecycle for function execution:

```go
package execution

import (
	"context"
	"fmt"
	"time"

	"yourproject/internal/containermanager"
	"yourproject/pkg/types"
)

// Executor manages the complete lifecycle of function execution
type Executor struct {
	containerManager containermanager.ContainerManager
	metrics          *types.Metrics
	config           ExecutorConfig
}

// ExecutorConfig holds configuration for the executor
type ExecutorConfig struct {
	DefaultTimeout    time.Duration
	MaxMemoryMB       int64
	MaxCPUQuota       int64
	TempDir           string
	BaseImages        map[string]string // runtime -> image
}

// NewExecutor creates a new function executor
func NewExecutor(cm containermanager.ContainerManager, metrics *types.Metrics, config ExecutorConfig) *Executor {
	// TODO 1: Validate config has required fields
	// TODO 2: Set defaults for missing values
	// TODO 3: Return initialized Executor
	return &Executor{
		containerManager: cm,
		metrics:          metrics,
		config:           config,
	}
}

// ExecuteFunction runs a function in an isolated container
func (e *Executor) ExecuteFunction(ctx context.Context, def *types.FunctionDefinition, req *types.InvocationRequest)
(*types.InvocationResponse, error) {
	// TODO 1: Validate function definition has required fields
	// TODO 2: Check if request has already expired (req.IsExpired())
```

```
        // TODO 3: Create container request from function definition:
        //         - Image: e.config.BaseImages[def.Runtime]
        //         - Cmd: language-specific command (e.g., ["/function", "handler"])
        //         - Env: combine def.EnvVars with system variables
        //         - MemoryLimit: def.MemoryMB * 1024 * 1024
        //         - CPUQuota: def.CPUQuota
        //         - Labels: map with function name, version, request ID
        //         - SecurityOpts: restrictive defaults
        // TODO 4: Call e.containerManager.CreateContainer()
        // TODO 5: Start container and record start time
        // TODO 6: Mount function artifact into container (/function directory)
        // TODO 7: Execute the handler with request payload as input
        // TODO 8: Capture output and record end time
        // TODO 9: Calculate duration, memory usage from stats
        // TODO 10: Build InvocationResponse with results
        // TODO 11: Always clean up container in defer
        // TODO 12: Return response and any execution error
        return nil, nil
}

// CleanupIdleContainers removes containers that have been idle beyond threshold
func (e *Executor) CleanupIdleContainers(ctx context.Context, idleThreshold time.Duration) error {
        // TODO 1: List all containers with function labels
        // TODO 2: For each container, check last used time from labels
        // TODO 3: If idle longer than threshold, stop and remove
        // TODO 4: Log cleanup actions
        // TODO 5: Return error only if cleanup fails catastrophically
        return nil
}
```

Language-Specific Hints

1. **Docker Go SDK**: Use `github.com/docker/docker/client` version `v20.10.17+incompatible` or later. Remember to close responses: `defer response.Body.Close()`.
2. **Resource Limits**: When setting CPU limits, Docker uses nanoseconds of CPU time per second. Convert percentage to nanoseconds: `CPUQuota = percentage * 10000` (e.g., 50% = 500000).
3. **Error Handling**: Docker API calls can fail with `ErrNotFound`, `ErrNotRunning`, etc. Use `client.IsErrNotFound(err)` to check error types.
4. **Context Propagation**: Always pass the context through to Docker API calls to support cancellation and timeouts.
5. **Security**: Generate seccomp profile JSON and pass as string: `SecurityOpt: []string{"seccomp=" + profileJSON}`.
6. **Metrics Integration**: Use the `types.Metrics` struct to record container operations: `e.metrics.ContainerOperations.WithLabelValues("create").Inc()`.

Milestone Checkpoint

After implementing the execution environment, verify it works:

1. **Test Container Creation**:

```bash
# Start the server
go run cmd/server/main.go

# In another terminal, test with curl
curl -X POST http://localhost:8080/v1/functions \
  -H "Content-Type: multipart/form-data" \
  -F "code=@hello.go" \
  -F "runtime=go" \
  -F "handler=Handler" \
  -F "name=test-func"

# Expected: Function created response with version ID
```

2. **Test Function Execution**:

```bash
# Invoke the function
curl -X POST http://localhost:8080/v1/functions/test-func/invoke \
  -H "Content-Type: application/json" \
  -d '{"name": "World"}'

# Expected: Response from function, check Docker containers are created
docker ps --filter "label=function=test-func"
```

3. **Verify Resource Limits**:

```bash
# Check container stats while function is running
CONTAINER_ID=$(docker ps -q --filter "label=function=test-func")
docker stats $CONTAINER_ID

# Should show memory limit enforced
```

4. **Test Cleanup**:

```bash
# Wait for function timeout + cleanup period
sleep 120

# Verify containers are removed
docker ps -a --filter "label=function=test-func"
# Expected: No containers (or only exited ones)
```

Signs of Problems:

- **"Cannot connect to Docker daemon"**: Docker isn't running or user lacks permissions
- **"No such image"**: Base images not pulled; run `docker pull` for runtime images
- **"Permission denied"**: Security options too restrictive; check seccomp profile
- **Containers not cleaned up**: Garbage collection not running; check cleanup timer

---

Component Design: Cold Start Optimization

> **Milestone(s):** Milestone 3

Mental Model: The Taxi Stand

Imagine you're managing a large taxi service in a busy city. There are two ways passengers get rides:

1. **The Taxi Stand Approach:** You maintain a fleet of taxis parked at strategic locations around the city. Each taxi is clean, fueled, and the engine is running. When a passenger arrives, they can immediately get into a waiting taxi and drive away. The taxi driver is already behind the wheel, knows the city layout, and is ready to go. The passenger's journey starts within seconds.
2. **The Custom-Built Approach:** When a passenger arrives, you call a factory to manufacture a new taxi from scratch. Workers assemble the chassis, install the engine, paint the exterior, install seats, and hire/train a driver. Only then can the passenger begin their journey. This process takes minutes, not seconds.

Cold starts in serverless functions are exactly this problem. The "custom-built approach" represents a true **cold start**: when a function hasn't been invoked recently, the system must:

- Provision a new execution environment (assemble the taxi)
- Install the runtime and dependencies (install the engine and seats)
- Load the function code (train the driver on the specific route)
- Execute the handler (begin the journey)

The **taxi stand approach** represents **warm start optimization**: we maintain a pool of pre-initialized, ready-to-go execution environments (warm containers). When a request arrives, we simply assign it to an available warm environment, eliminating most of the startup overhead.

The key insight is that we're trading **resource efficiency** (keeping idle instances ready consumes memory and CPU) for **performance** (sub-100ms response times). Our design must intelligently balance this trade-off by determining how many taxis to keep idling, where to park them, and when to send them back to the factory for recycling.

Warm Pool Manager Interface

The **Warm Pool Manager** is the component responsible for maintaining a pool of pre-initialized `FunctionInstance` objects and allocating them to incoming requests. It serves as the intermediary between the **Request Router** (which needs execution environments) and the **Container Manager** (which creates and destroys them). Think of it as the dispatcher at the taxi stand who tracks which taxis are available, which are currently on trips, and decides when to call for more taxis from the factory.

The manager operates on two key principles:

1. **Instance Reuse:** Once a function finishes executing, its container environment is returned to the warm pool (after cleanup) rather than destroyed, ready for the next invocation of the same function.
2. **Pre-warming:** Based on traffic patterns, the system can proactively create and initialize instances before they're needed, anticipating demand spikes.

Here are the core data structures and interfaces:

Type/Interface	Purpose
`WarmPoolManager`	Main interface for managing warm instances
`PoolConfig`	Configuration for pool behavior and limits
`AcquireRequest`	Request to obtain a warm instance
`PoolMetrics`	Metrics tracking pool efficiency and performance

Pool Configuration:

Field	Type	Description
`MaxWarmInstances`	`int`	Maximum number of warm instances to keep per function (prevents memory bloat)
`MinWarmInstances`	`int`	Minimum number of warm instances to maintain per function (for latency-sensitive workloads)
`InstanceTTL`	`time.Duration`	Maximum lifetime of a warm instance before it's recycled (prevents stale environments)
`IdleTimeout`	`time.Duration`	How long an instance can sit idle in the warm pool before being terminated (scale-to-zero within pool)
`PreWarmThreshold`	`float64`	Request rate threshold that triggers pre-warming (requests per second)
`CleanupInterval`	`time.Duration`	How often to run cleanup of expired/stale instances

Warm Pool Manager Interface:

Method	Parameters	Returns	Description
`AcquireWarmInstance`	`ctx context.Context`, `functionName string`, `version string`	`(*FunctionInstance, error)`	Gets a warm instance for the specified function. Creates one if needed and pool limits allow.
`ReleaseInstance`	`ctx context.Context`, `instance *FunctionInstance`, `healthy bool`	`error`	Returns an instance to the pool after execution. If `healthy` is false, the instance is destroyed instead.
`PreWarm`	`ctx context.Context`, `functionName string`, `version string`, `count int`	`error`	Proactively creates `count` warm instances for the function.
`DrainPool`	`ctx context.Context`, `functionName string`, `version string`	`error`	Terminates all warm instances for a function (used during updates or scale-to-zero).
`GetPoolStats`	`ctx context.Context`, `functionName string`, `version string`	`(*PoolStats, error)`	Returns current pool statistics (available instances, total instances, etc.).
`CleanupExpired`	`ctx context.Context`	`error`	Removes instances that have exceeded their TTL or idle timeout.

Instance States with Pool Transitions: The `FunctionInstanceState` machine from our data model expands with pool-specific transitions:

Current State	Event	Next State	Action Taken
`InstanceStateWarm`	`AcquireWarmInstance` called	`InstanceStateActive`	Instance removed from available pool, assigned to request
`InstanceStateActive`	Execution completes	`InstanceStateWarm`	Instance cleaned up (filesystem reset), returned to warm pool
`InstanceStateWarm`	`IdleTimeout` expires	`InstanceStateDraining`	Begin graceful termination (finish any in-progress operations)
`InstanceStateWarm`	`InstanceTTL` expires	`InstanceStateDraining`	Begin graceful termination (instance is stale)
`InstanceStateWarm`	Function version updated	`InstanceStateDraining`	Old version instances are drained during updates
Any state	Container health check fails	`InstanceStateError`	Instance marked unhealthy, removed from pool, replacement created

> **Design Insight:** The warm pool isn't just about keeping containers running—it's about keeping *initialized* containers. True cold start optimization requires that the runtime environment (language interpreter, loaded libraries, framework initialization) is already prepared. Our warm instances are "pre-heated" execution environments with the function code loaded and the handler initialized, ready to process the first instruction immediately.

ADR: Pooling Strategy - Eager vs. Lazy

> **Decision: Lazy Warm Pool with Predictive Top-Ups**
>
> - **Context:** We need to minimize cold starts while avoiding excessive resource consumption. Keeping thousands of instances warm "just in case" would waste memory and CPU, especially for infrequently invoked functions. However, waiting for a cold start on every invocation defeats the purpose of optimization.
> - **Options Considered:**
>   1. **Eager Pooling (Fixed-Size):** Maintain a fixed number of warm instances per function at all times, regardless of traffic.
>   2. **Lazy Pooling (On-Demand):** Create warm instances only when a request arrives and no warm instance is available.
>   3. **Hybrid Predictive:** Use lazy pooling as baseline, but proactively create instances when traffic patterns suggest increased demand.
> - **Decision:** We choose **Hybrid Predictive** with lazy creation as the default behavior.
> - **Rationale:**
>   - **Eager pooling** wastes resources for low-traffic functions (keeping instances warm for hours between invocations).
>   - **Pure lazy pooling** still incurs cold start latency for the first request after idle periods.
>   - **Hybrid predictive** gives us the best of both: minimal resource usage during quiet periods, but reduced latency when traffic patterns change. By monitoring request rates and using simple heuristics (like "if requests-per-second increases by X% over Y seconds, pre-warm Z instances"), we can anticipate demand without complex ML models.
> - **Consequences:**
>   - **Positive:** Good balance of resource efficiency and performance, adaptable to changing workloads.
>   - **Negative:** Requires implementing traffic pattern analysis and prediction logic, adds complexity.
>   - **Trade-off:** Some cold starts still occur for truly unpredictable traffic spikes, but this is acceptable for most workloads.

Options Comparison:

Option	Pros	Cons	Why Not Chosen
Eager Pooling	- Guaranteed zero cold starts for first N concurrent requests - Simple implementation (static pool)	- Wastes resources for infrequent functions - Doesn't adapt to traffic patterns - Higher memory footprint increases hosting costs	Too resource-inefficient for a multi-tenant system where most functions are invoked infrequently
Lazy Pooling	- Optimal resource usage (zero idle instances) - Simple to implement	- Every scale-up event incurs cold start latency - Poor user experience during traffic spikes - Could cause request queuing during sudden load	Unacceptable latency during traffic growth; violates performance goals
Hybrid Predictive	- Balances resources and performance - Adapts to workload patterns - Can be tuned based on function importance/cost	- More complex implementation - Requires metrics collection and analysis - Prediction can be wrong (false positives waste resources)	**CHOSEN:** Provides the best trade-off for our target workloads; complexity is manageable

Common Pitfalls in Cold Start Optimization

⚠️ Pitfall: Memory Bloat from Unlimited Warm Pools

The Mistake: Keeping an unlimited number of warm instances per function, or setting `MaxWarmInstances` too high. Each warm instance consumes memory (the container's resident memory plus any loaded libraries). If you keep 100 instances warm for a function that uses 256MB each, that's 25GB of RAM occupied—even if the function only gets invoked once per hour.

Why It's Wrong: This wastes expensive memory resources and reduces overall system density. In a multi-tenant environment, one greedy function could starve others of resources. It also increases costs for the platform operator.

How to Fix: Implement strict per-function limits (`MaxWarmInstances`) based on function tier/plan. Use metrics to monitor pool memory consumption. Implement **instance eviction** policies: when at capacity, evict the least-recently-used warm instance to make room for new ones.

⚠️ Pitfall: Stale Environment State Between Invocations

The Mistake: Returning a container to the warm pool without properly resetting its state. For example, if a function writes files to `/tmp` or modifies environment variables, those changes persist to the next invocation, potentially leaking data between users or causing unexpected behavior.

Why It's Wrong: Violates the principle of isolation between invocations. User A's data could be visible to User B, creating security vulnerabilities. Stateful behavior makes debugging difficult and breaks the "stateless" assumption of serverless functions.

How to Fix: Implement a **cleanup hook** before returning an instance to the warm pool:

1. Delete all files in the container's writable filesystem (especially `/tmp`)
2. Reset environment variables to their original values
3. Clear any in-memory caches (language runtime specific)
4. Restart the container's entrypoint process (if possible) to ensure fresh runtime state Consider using copy-on-write filesystem layers to efficiently create fresh filesystem state for each invocation.

⚠️ Pitfall: Over-Complicated Snapshot/Restore Mechanisms

The Mistake: Attempting to implement full container checkpoint/restore (CRIU) or VM snapshotting without understanding the complexity and edge cases. These technologies promise near-instant restores but come with significant limitations: large snapshot files, compatibility issues with certain syscalls, and high implementation complexity.

Why It's Wrong: The development time and maintenance burden often outweigh the benefits. Snapshot files can be large (hundreds of MB), requiring significant storage I/O that might negate latency gains. CRIU has limitations with multi-threaded applications, certain kernel versions, and network connections.

How to Fix: Start with **simple warm pooling** (keeping containers running), which provides 80% of the benefit with 20% of the complexity. Only consider snapshot/restore for specific use cases after measuring actual cold start bottlenecks. If implemented, use it selectively for large runtimes (like JVM) where initialization is truly expensive, not for lightweight runtimes like Go.

⚠️ Pitfall: Ignoring Language-Specific Initialization Costs

The Mistake: Treating all runtime environments equally. A pre-initialized Node.js container might start in 50ms, while a JVM container with Spring Boot could take 10 seconds to initialize classes and dependencies.

Why It's Wrong: Different runtimes have vastly different cold start characteristics. A one-size-fits-all pooling strategy will either over-provision (wasting resources on fast runtimes) or under-provision (causing unacceptable latency for slow runtimes).

How to Fix: Implement **runtime-aware pooling**:

- Set different `MinWarmInstances` based on runtime (higher for JVM, lower for Go)
- Use runtime-specific optimizations: for JVM, consider keeping instances in "warm" state with classpath pre-loaded but no application context initialized
- Implement **gradual warmup**: initialize heavy frameworks in the background while serving simple requests
- Track cold start duration per runtime in metrics and adjust pooling parameters accordingly

Implementation Guidance for Cold Start

Technology Recommendations:

Component	Simple Option	Advanced Option
Pool Storage	In-memory map with mutex synchronization	Redis with distributed locking for multi-node deployments
Instance Cleanup	Basic filesystem wipe (`os.RemoveAll`)	Overlay filesystem with fresh writable layer per invocation
Metrics Collection	Prometheus counters and gauges	OpenTelemetry with distributed tracing for end-to-end latency analysis
Predictive Warming	Simple rate-of-change threshold	Machine learning model trained on historical invocation patterns

Recommended File/Module Structure:

```
project-root/
  cmd/
    server/main.go                  # Entry point
  internal/
    warmpool/                       # Warm pool component
      manager.go                    # WarmPoolManager implementation
      config.go                     # PoolConfig and related types
      metrics.go                    # Pool-specific metrics
      cleanup.go                    # Instance cleanup utilities
      poolstore/                    # Storage backend for pool state
        memory_store.go             # In-memory implementation
        redis_store.go              # Redis implementation (optional)
    container/                      # Container management (from Milestone 2)
      manager.go                    # ContainerManager interface
      docker_manager.go             # Docker implementation
    executor/                       # Function execution (uses warm pool)
      executor.go                   # Executor that acquires/releases instances
    gateway/                        # HTTP gateway (from Milestone 4)
      router.go                     # Request routing
    scaling/                        # Auto-scaling (Milestone 5)
      scaler.go                     # Uses pool metrics for decisions
```

Infrastructure Starter Code - Warm Pool Manager Base Implementation:

GO

```go
// internal/warmpool/manager.go

package warmpool

import (
    "context"
    "fmt"
    "sync"
    "time"

    "github.com/prometheus/client_golang/prometheus"
    "project/internal/types"
    "project/internal/container"
)

var (
    // Pool metrics
    warmPoolSize = prometheus.NewGaugeVec(
        prometheus.GaugeOpts{
            Name: "warm_pool_size",
            Help: "Number of instances in warm pool per function",
        },
        []string{"function", "runtime"},
    )
    coldStarts = prometheus.NewCounterVec(
        prometheus.CounterOpts{
            Name: "cold_starts_total",
            Help: "Total number of cold starts",
        },
        []string{"function", "runtime"},
    )
    poolHits = prometheus.NewCounterVec(
        prometheus.CounterOpts{
            Name: "warm_pool_hits_total",
            Help: "Number of times warm instance was available",
        },
        []string{"function", "runtime"},
    )
    poolMisses = prometheus.NewCounterVec(
        prometheus.CounterOpts{
            Name: "warm_pool_misses_total",
            Help: "Number of times warm instance was not available",
```

```
        },
        []string{"function", "runtime"},
    )
)

func init() {
    prometheus.MustRegister(warmPoolSize, coldStarts, poolHits, poolMisses)
}

// PoolConfig holds configuration for warm pool behavior
type PoolConfig struct {
    MaxWarmInstances   int           // Maximum warm instances per function
    MinWarmInstances   int           // Minimum warm instances to maintain
    InstanceTTL        time.Duration // Max lifetime of a warm instance
    IdleTimeout        time.Duration // Time before idle instance is terminated
    CleanupInterval    time.Duration // How often to run cleanup
    PreWarmThreshold   float64       // RPS threshold for pre-warming
}

// WarmPoolManager manages a pool of warm function instances
type WarmPoolManager struct {
    containerManager container.ContainerManager
    config           PoolConfig
    store            PoolStore
    metrics          *types.Metrics
    cleanupTicker    *time.Ticker
    mu               sync.RWMutex

    // Function definitions cache for creating new instances
    functionDefs map[string]*types.FunctionDefinition
}

// NewWarmPoolManager creates a new warm pool manager
func NewWarmPoolManager(cm container.ContainerManager, config PoolConfig, metrics *types.Metrics) *WarmPoolManager {
    wpm := &WarmPoolManager{
        containerManager: cm,
        config:           config,
        store:            NewMemoryStore(),
        metrics:          metrics,
        functionDefs:     make(map[string]*types.FunctionDefinition),
        cleanupTicker:    time.NewTicker(config.CleanupInterval),
    }
```

```go
    // Start background cleanup goroutine
    go wpm.cleanupLoop()

    return wpm
}

// AcquireWarmInstance gets a warm instance for a function
func (wpm *WarmPoolManager) AcquireWarmInstance(ctx context.Context, functionName, version string) (*types.FunctionInstance, error) {
    // TODO 1: Generate a composite key from functionName and version
    // TODO 2: Check if a warm instance is available in the store
    // TODO 3: If available: remove from pool, update metrics (poolHits), update instance state to ACTIVE, return it
    // TODO 4: If not available: update metrics (poolMisses, coldStarts)
    // TODO 5: Check if we're below MaxWarmInstances limit for this function
    // TODO 6: If under limit: create a new instance (cold start), set state to ACTIVE, return it
    // TODO 7: If at limit: wait for an instance to become available (with timeout) or return error
    return nil, fmt.Errorf("not implemented")
}

// ReleaseInstance returns an instance to the pool after execution
func (wpm *WarmPoolManager) ReleaseInstance(ctx context.Context, instance *types.FunctionInstance, healthy bool) error {
    // TODO 1: If instance is not healthy: destroy it completely, return nil
    // TODO 2: Clean up the instance: reset filesystem, clear environment state
    // TODO 3: Check if instance has exceeded InstanceTTL
    // TODO 4: If exceeded TTL: destroy it, return nil
    // TODO 5: Update instance state to WARM, set LastUsedAt to now
    // TODO 6: Add instance back to the pool store
    // TODO 7: Update warmPoolSize metrics
    return fmt.Errorf("not implemented")
}

// PreWarm proactively creates warm instances
func (wpm *WarmPoolManager) PreWarm(ctx context.Context, functionName, version string, count int) error {
    // TODO 1: Get current warm instance count for this function
    // TODO 2: Calculate how many new instances to create (respecting MaxWarmInstances)
    // TODO 3: For each instance to create: launch async creation goroutine
    // TODO 4: Each goroutine: create container, initialize it, add to warm pool
    // TODO 5: Return early (don't wait for all instances to be ready)
    return fmt.Errorf("not implemented")
}

// cleanupLoop runs periodic cleanup of expired instances
```

```go
func (wpm *WarmPoolManager) cleanupLoop() {

    for range wpm.cleanupTicker.C {

        wpm.cleanupExpired(context.Background())

    }

}


func (wpm *WarmPoolManager) cleanupExpired(ctx context.Context) error {

    // TODO 1: Iterate through all instances in the store

    // TODO 2: For each instance: check if IdleTimeout has expired

    // TODO 3: If expired: remove from pool, destroy container

    // TODO 4: Update warmPoolSize metrics

    return nil

}
```

Core Logic Skeleton Code - Instance Cleanup Hook:

```go
// internal/warmpool/cleanup.go

package warmpool


import (

    "context"

    "os"

    "path/filepath"


    "project/internal/container"

    "project/internal/types"

)


// CleanupInstance resets an instance to a clean state before returning to pool

func CleanupInstance(ctx context.Context, cm container.ContainerManager, instance *types.FunctionInstance) error {

    // TODO 1: Clean filesystem - execute command in container to delete all files in /tmp

    // Hint: Use cm.ExecuteInContainer with command: ["sh", "-c", "rm -rf /tmp/*"]


    // TODO 2: Reset environment variables - restart the container's entrypoint process

    // Hint: For Docker: stop and start the container (or send signal to restart process)


    // TODO 3: Clear runtime-specific caches

    // Hint: Detect runtime from instance and apply runtime-specific cleanup

    // - Node.js: Clear require.cache? (difficult in separate process)

    // - Python: Might need to restart Python process entirely

    // - JVM: Consider keeping JVM warm but reset application context


    // TODO 4: Verify container health (run a simple echo command)

    // TODO 5: If any cleanup step fails, mark instance as unhealthy


    return nil

}


// Runtime-specific cleanup strategies

type RuntimeCleaner interface {

    Clean(ctx context.Context, containerID string) error

}


func GetCleanerForRuntime(runtime string) RuntimeCleaner {

    switch runtime {

    case "go":

        return &GoCleaner{}

    case "python":
```

```
        return &PythonCleaner{}
    case "node":
        return &NodeJSCleaner{}
    case "java":
        return &JavaCleaner{}
    default:
        return &GenericCleaner{}
    }
}
```

Integration with Executor - Modified ExecuteFunction Method:

```go
// internal/executor/executor.go - Modified from Milestone 2

func (e *Executor) ExecuteFunction(ctx context.Context, def *types.FunctionDefinition, req *types.InvocationRequest)
(*types.InvocationResponse, error) {

    // TODO 1: Acquire warm instance from pool (instead of creating new container)

    instance, err := e.warmPool.AcquireWarmInstance(ctx, def.Name, req.FunctionVersion)

    if err != nil {

        return nil, fmt.Errorf("failed to acquire instance: %w", err)

    }



    // TODO 2: If instance is nil (pool empty and at max capacity), wait or return error



    // TODO 3: Execute function in the acquired container

    // ... existing execution logic from Milestone 2 ...



    // TODO 4: After execution completes, release instance back to pool

    defer func() {

        // Pass true if execution succeeded, false if container should be destroyed

        healthy := (err == nil)

        releaseErr := e.warmPool.ReleaseInstance(ctx, instance, healthy)

        if releaseErr != nil {

            // Log error but don't fail the invocation

            e.metrics.ContainerOperations.WithLabelValues("release_failed").Inc()

        }

    }()



    // ... rest of execution ...

}
```

Language-Specific Hints for Go:

1. **Concurrency Management:** Use `sync.RWMutex` for pool store access—multiple goroutines can read concurrently, but only one can write. For the pool store map: `map[string][]*FunctionInstance`.

2. **Background Goroutines:** Use `context.Context` to gracefully stop the cleanup goroutine on shutdown:

```go
go func() {
    for {
        select {
        case <-ctx.Done():
            return
        case <-wpm.cleanupTicker.C:
            wpm.cleanupExpired(ctx)
        }
    }
}()
```

3. **Metrics Collection:** Use Prometheus histogram for cold start duration:

```go
coldStartDuration := prometheus.NewHistogramVec(
    prometheus.HistogramOpts{
        Name: "cold_start_duration_ms",
        Help: "Duration of cold starts in milliseconds",
        Buckets: []float64{10, 50, 100, 200, 500, 1000, 5000},
    },
    []string{"function", "runtime"},
)
```

4. **Instance Identification:** Create a unique key for each function version:

```go
func poolKey(functionName, version string) string {
    return fmt.Sprintf("%s:%s", functionName, version)
}
```

Milestone Checkpoint Verification:

After implementing the warm pool, verify the following behaviors:

1. **Cold Start Reduction:**

```
# Send two requests sequentially with a small delay
curl -X POST http://localhost:8080/functions/myfunc
# First request should show "cold_start: true" in logs or metrics

curl -X POST http://localhost:8080/functions/myfunc
# Second request should show "cold_start: false" (warm start)
```

2. **Pool Metrics Exposure:**

```
curl http://localhost:9090/metrics | grep warm_pool
# Should see:
# warm_pool_size{function="myfunc",runtime="go"} 1
# warm_pool_hits_total{function="myfunc",runtime="go"} 1
```

3. **Instance Cleanup Verification:**

```
# Check container count before and after idle timeout
docker ps | grep myfunc
# Wait for IdleTimeout duration (e.g., 5 minutes)
# Check again - idle instances should be removed
```

4. **Concurrent Request Handling:**

```
# Send 10 concurrent requests
for i in {1..10}; do
  curl -X POST http://localhost:8080/functions/myfunc &
done
# Should see multiple warm instances created (up to MaxWarmInstances)
# Check warm_pool_size metric
```

Common Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
All requests show cold starts	Warm pool not being populated	Check if `ReleaseInstance` is being called after execution	Ensure executor calls `ReleaseInstance` in `defer`
Memory usage grows indefinitely	Instances not being cleaned up	Check `cleanupExpired` logs; verify `IdleTimeout` is > 0	Ensure cleanup goroutine is running; check for deadlocks
Instance reuse causes state leakage	Cleanup not working properly	Add debug logs in `CleanupInstance`; check `/tmp` contents in container	Ensure cleanup commands execute successfully in container
High latency on first request after idle	`MinWarmInstances` set to 0	Check pool size metrics during idle period	Increase `MinWarmInstances` or implement predictive warming
"Too many warm instances" errors	`MaxWarmInstances` too low for concurrent load	Monitor concurrent request rate vs pool size	Increase `MaxWarmInstances` or implement request queuing

---

Component Design: Request Routing

> **Milestone(s):** Milestone 4

The request routing component serves as the system's **front door**—it receives incoming function invocation requests, determines which function instance should handle each request, manages request flow when all instances are busy, and ensures reliable delivery even during scaling events and failures. This component bridges external HTTP traffic with the internal execution infrastructure, making critical decisions about load distribution, timeout handling, and concurrency management.

Mental Model: The Restaurant Host and Waitlist

Imagine a popular restaurant with multiple identical dining rooms (function instances). When customers (requests) arrive, they're greeted by a host (gateway) who must decide:

1. **Which dining room to seat them in**: The host looks at currently available rooms (warm instances) that match the requested cuisine type (function). If multiple rooms are available, the host uses a strategy (load balancing) to distribute customers evenly.
2. **What to do when all rooms are full**: The host maintains a waitlist (request queue) where new arrivals wait for the next available room. The waitlist has limited capacity to prevent overcrowding in the entrance area (memory exhaustion).
3. **When to turn customers away**: If the waitlist is full, or if customers have been waiting too long (timeout), the host politely informs them they cannot be served today (returns an error).
4. **How to handle special requests**: Some customers request a specific room they've used before (sticky routing for stateful functions), while others don't care. Some customers want to wait for their meal (synchronous invocation), while others leave their phone number and want to be called when it's ready (asynchronous invocation).

This mental model captures the core responsibilities of request routing: **matching**, **queueing**, **timeout management**, and **load distribution** while maintaining system stability under varying load.

Gateway API and Internal Routing

The gateway exposes a simple HTTP API that accepts function invocation requests and routes them to appropriate function instances. Internally, it implements a sophisticated routing engine that considers instance availability, load, and health status.

Public HTTP Gateway API

The gateway accepts HTTP POST requests at `/invoke/{functionName}` with the following characteristics:

Request Component	Specification	Purpose
URL Path	`/invoke/{functionName}[:{versionTag}]`	Identifies the target function and optional version (default: latest)
HTTP Method	POST	All invocation types use POST to include payload
Headers	`X-Function-Timeout` : Override default timeout (milliseconds) `X-Synchronous` : "true"/"false" (default: true) `X-Callback-URL` : For async invocations `X-Request-ID` : Client-provided correlation ID	Control invocation behavior and metadata
Body	Arbitrary bytes (content-type preserved)	Function input payload
Query Params	`async=true` , `timeout=5000` , `version=v1.2`	Alternative to headers for simpler clients

The gateway returns:

- **200 OK** with function output in body for synchronous invocations
- **202 Accepted** with `X-Request-ID` for asynchronous invocations
- **429 Too Many Requests** when request queue is full
- **503 Service Unavailable** when function is scaling or unhealthy
- **504 Gateway Timeout** when function execution exceeds timeout

Internal Routing Mechanism

When a request arrives, the gateway follows this routing pipeline:

1. **Request Validation**: Parse and validate the function name, version, timeout, and payload size.
2. **Function Resolution**: Resolve the function name to a specific `FunctionDefinition` and check if it's active.
3. **Instance Selection**: Use the load balancing strategy to select an appropriate `FunctionInstance` .
4. **Queue Management**: If no instances are available, attempt to queue the request (if synchronous and queue not full).
5. **Request Forwarding**: Forward the request to the selected instance via HTTP or gRPC.
6. **Response Handling**: Collect the response, apply any transformations, and return to client.
7. **Error Recovery**: Handle instance failures with retries (when appropriate) and circuit breaking.

The internal routing table maintains a mapping from function name to available instances:

Routing Table Entry Field	Type	Description
`FunctionName`	`string`	Unique function identifier
`ActiveInstances`	`[]*FunctionInstance`	Instances in `InstanceStateWarm` or `InstanceStateActive` state
`PendingRequests`	`*RequestQueue`	Queue of requests waiting for instances
`ConcurrentInvocations`	`int`	Current number of in-flight requests
`MaxConcurrency`	`int`	Maximum concurrent executions allowed
`CircuitBreaker`	`*CircuitBreaker`	Tracks instance health for fail-fast behavior
`LastScaleUpTime`	`time.Time`	When last scale-up occurred (for cooldown)

Instance Selection Algorithm

The instance selection process follows these steps:

1. **Filter by state**: Only consider instances with `IsActive() == true` (typically `InstanceStateWarm` or `InstanceStateActive` but not `InstanceStateDraining`).
2. **Apply stickiness**: If `InvocationRequest.PreferredInstanceID` is set and that instance is active, use it (for stateful functions).
3. **Check circuit breakers**: Skip instances marked as unhealthy (recent failures exceed threshold).
4. **Apply load balancing strategy**: Use the configured strategy (round-robin, least-connections, etc.) to select from remaining instances.
5. **Fallback to queuing**: If no instances available, attempt to queue (for synchronous requests) or immediately fail (for async if no retry capacity).

Request Queue Design

The request queue buffers incoming requests when all instances are busy. It's implemented as a bounded priority queue:

Queue Characteristic	Specification	Rationale
Maximum Size	Configurable per function (default: 100)	Prevents memory exhaustion during traffic spikes
Priority Scheme	FIFO with optional priority based on timeout	Ensures fairness; requests with earlier deadlines get priority
Eviction Policy	Oldest requests evicted when queue full (configurable)	Better to fail fast than stall entire system
Queue Drain	On scale-up, drain queue to new instances	Utilizes new capacity immediately
Monitoring	Queue depth, wait time, eviction count metrics	Essential for capacity planning

ADR: Load Balancing Strategy

Decision: Least-Connections with Circuit Breaking

- **Context**: We need to distribute requests across multiple function instances to maximize throughput, minimize latency, and handle instance failures gracefully. The strategy must work with auto-scaling (instances come and go) and support both stateless and stateful functions.
- **Options Considered**:
  1. **Round-Robin**: Simple rotation through available instances
  2. **Least-Connections**: Send to instance with fewest active requests
  3. **Consistent Hashing**: Map requests to specific instances based on request attributes
- **Decision**: Least-connections as the default, with optional consistent hashing for stateful functions via `PreferredInstanceID`.
- **Rationale**: Least-connections naturally balances load when instances have variable execution times (common in serverless). It prevents overloading slow instances and works well with auto-scaling as new instances start with zero connections. Round-robin can overload instances processing long requests. Consistent hashing adds complexity but is valuable for stateful functions, which we support through explicit instance preference rather than automatic hashing.
- **Consequences**: Requires tracking active invocation count per instance (already needed for scaling). Adds slight overhead vs. round-robin but provides better load distribution. Stateful functions need client cooperation to specify preferred instance.

Option	Pros	Cons	Chosen?
Round-Robin	Simple, predictable, no state needed	Can overload instances with long-running requests, doesn't account for instance capacity differences	No (fallback when connection counts unavailable)
Least-Connections	Adapts to instance performance variations, naturally balances load, works well with auto-scaling	Requires tracking per-instance invocation counts, slightly more complex	**Yes** (default strategy)
Consistent Hashing	Enables sticky sessions for stateful functions, good cache locality	Complex implementation, rehashing on scale events disrupts stickiness, uneven load distribution	Partially (via `PreferredInstanceID` for explicit stickiness)

Common Pitfalls in Request Routing

⚠️ **Pitfall: Unbounded Queue Growth**

- **Description**: Allowing request queues to grow without limits during traffic spikes.
- **Why it's wrong**: Exhausts system memory, increases latency unpredictably, and can cause cascading failures when queues consume all available RAM.
- **How to avoid**: Implement **bounded queues** with configurable maximum size per function. Monitor queue depth and implement queue pressure signals to upstream clients (HTTP 429). Consider multiple queue tiers: in-memory for short bursts, disk-backed for important async workloads.

⚠️ **Pitfall: Thundering Herd on Scale-Up**

- **Description**: When a function scales from zero, all queued requests simultaneously rush to the single new instance, overwhelming it.
- **Why it's wrong**: Causes immediate timeout or failure of the new instance, triggering unhealthy marks and potential scale oscillation.
- **How to avoid**: Implement **gradual queue draining** - release requests to new instances at a controlled rate (e.g., max N per second). Use **per-instance concurrency limits** to prevent overloading single instances. Consider **delayed scaling** where you wait for multiple instances before draining queue.

⚠️ **Pitfall: Improper Timeout Handling**

- **Description**: Not propagating or respecting timeouts correctly through the routing chain.
- **Why it's wrong**: Client receives timeout after function continues executing (wasting resources), or function times out before client deadline causing unnecessary retries.
- **How to avoid**: Implement **timeout deadline propagation** from client request through all routing layers to function execution. Use a **timeout hierarchy**: client timeout > gateway timeout > function timeout. Always cancel downstream operations when upstream timeout occurs.

⚠️ **Pitfall: Ignoring Cold Start Latency in Routing Decisions**

- **Description**: Routing requests to cold instances when warm ones are available, or not accounting for cold start time in queue wait calculations.
- **Why it's wrong**: Increases tail latency unpredictably, especially for low-traffic functions.
- **How to avoid**: Track instance **warm/cold status** and prefer warm instances. Adjust queue timeout calculations to include estimated cold start time. Implement **predictive pre-warming** based on queue depth and historical patterns.

⚠️ **Pitfall: No Circuit Breaking for Unhealthy Instances**

- **Description**: Continuing to route requests to instances that are failing or slow.
- **Why it's wrong**: Degrades overall system performance, amplifies failures, and creates negative user experience.
- **How to avoid**: Implement **circuit breaker pattern** per instance: track failure rates and temporarily remove instances from rotation after threshold breaches. Combine with health checks for proactive failure detection. Allow **gradual recovery** (probation period) for healed instances.

Implementation Guidance for Routing

Technology Recommendations Table

Component	Simple Option	Advanced Option	Recommendation
HTTP Gateway	Go `net/http` with middleware	`gorilla/mux` for routing, `chi` for middleware	`net/http` with custom routing (keeps dependencies minimal)
Load Balancer	In-memory round-robin	Distributed consistent hashing with rendezvous hashing	Least-connections with atomic counters
Request Queue	Go channel with buffer	Disk-backed queue (Redis, Kafka) for persistence	Bounded channel queue with priority via heap
Circuit Breaker	Simple counter with timeout	Adaptive breaker with rolling windows	`github.com/sony/gobreaker` or similar
Metrics	Prometheus counters/gauges	Distributed tracing (OpenTelemetry)	Prometheus for metrics, context propagation for traces

Recommended File/Module Structure

```
project-root/
  cmd/
    gateway/                    # Gateway entry point
      main.go                   # Starts HTTP gateway server
  internal/
    gateway/                    # Request routing component
      gateway.go                # Main gateway struct and HTTP handlers
      router.go                 # Routing logic and load balancing
      queue/                    # Request queue implementation
        queue.go                # Bounded priority queue
        manager.go              # Queue lifecycle management
      circuit/                  # Circuit breaker implementation
        breaker.go              # Circuit breaker logic
        health_check.go         # Instance health checks
      middleware/               # HTTP middleware
        logging.go              # Request logging
        metrics.go              # Prometheus metrics
        timeout.go              # Deadline propagation
      api/                      # Gateway API definitions
        models.go               # Request/response types
        errors.go               # Gateway-specific errors
    controller/                 # (Referenced, not in this component)
    worker/                     # (Referenced, not in this component)
    types/                      # Shared types (already defined)
```

Infrastructure Starter Code

Complete HTTP Gateway Server (ready to use):

```go
// internal/gateway/gateway.go

package gateway

import (
    "context"
    "encoding/json"
    "fmt"
    "net/http"
    "strconv"
    "time"

    "github.com/yourproject/internal/types"
    "github.com/prometheus/client_golang/prometheus"
    "github.com/prometheus/client_golang/prometheus/promhttp"
)

// Gateway is the main HTTP gateway for function invocations
type Gateway struct {
    router         *Router
    queueManager   *queue.Manager
    metrics        *gatewayMetrics
    config         GatewayConfig
    server         *http.Server
}

// GatewayConfig holds gateway configuration
type GatewayConfig struct {
    Addr                string
    MaxRequestBodyBytes  int64
    DefaultTimeout       time.Duration
    EnableMetrics        bool
    MetricsPath          string
    HealthCheckPath      string
}

// gatewayMetrics holds Prometheus metrics for the gateway
type gatewayMetrics struct {
    requestsTotal      *prometheus.CounterVec
    requestDuration    *prometheus.HistogramVec
    queueDepth         *prometheus.GaugeVec
    activeInvocations  prometheus.Gauge
    errorsTotal        *prometheus.CounterVec
```

```
}

// NewGateway creates a new gateway instance
func NewGateway(router *Router, qm *queue.Manager, config GatewayConfig) *Gateway {
    gm := &gatewayMetrics{
        requestsTotal: prometheus.NewCounterVec(
            prometheus.CounterOpts{
                Name: "gateway_requests_total",
                Help: "Total number of HTTP requests processed",
            },
            []string{"function", "method", "status"},
        ),
        requestDuration: prometheus.NewHistogramVec(
            prometheus.HistogramOpts{
                Name:    "gateway_request_duration_seconds",
                Help:    "HTTP request duration in seconds",
                Buckets: prometheus.DefBuckets,
            },
            []string{"function", "method"},
        ),
        queueDepth: prometheus.NewGaugeVec(
            prometheus.GaugeOpts{
                Name: "gateway_queue_depth",
                Help: "Current depth of request queues",
            },
            []string{"function"},
        ),
        activeInvocations: prometheus.NewGauge(
            prometheus.GaugeOpts{
                Name: "gateway_active_invocations",
                Help: "Number of currently active function invocations",
            },
        ),
        errorsTotal: prometheus.NewCounterVec(
            prometheus.CounterOpts{
                Name: "gateway_errors_total",
                Help: "Total number of gateway errors by type",
            },
            []string{"function", "error_type"},
        ),
    }
```

```
    prometheus.MustRegister(
        gm.requestsTotal,
        gm.requestDuration,
        gm.queueDepth,
        gm.activeInvocations,
        gm.errorsTotal,
    )

    return &Gateway{
        router:       router,
        queueManager: qm,
        metrics:      gm,
        config:       config,
    }
}

// Start begins serving HTTP requests
func (g *Gateway) Start() error {
    mux := http.NewServeMux()

    // Main invocation endpoint
    mux.HandleFunc("/invoke/", g.invokeHandler)

    // Health check endpoint
    mux.HandleFunc(g.config.HealthCheckPath, func(w http.ResponseWriter, r *http.Request) {
        w.WriteHeader(http.StatusOK)
        w.Write([]byte("OK"))
    })

    // Metrics endpoint (optional)
    if g.config.EnableMetrics {
        mux.Handle(g.config.MetricsPath, promhttp.Handler())
    }

    // Apply middleware chain
    handler := g.applyMiddleware(mux)

    g.server = &http.Server{
        Addr:         g.config.Addr,
        Handler:      handler,
```

```
        ReadTimeout:  30 * time.Second,
        WriteTimeout: 30 * time.Second,
    }

    return g.server.ListenAndServe()
}

// Shutdown gracefully stops the gateway
func (g *Gateway) Shutdown(ctx context.Context) error {
    return g.server.Shutdown(ctx)
}

// applyMiddleware chains middleware in correct order
func (g *Gateway) applyMiddleware(h http.Handler) http.Handler {
    // Order matters: last added = first executed
    h = timeoutMiddleware(g.config.DefaultTimeout)(h)
    h = metricsMiddleware(g.metrics)(h)
    h = loggingMiddleware()(h)
    h = recoveryMiddleware()(h)
    h = sizeLimitMiddleware(g.config.MaxRequestBodyBytes)(h)
    return h
}

// invokeHandler handles /invoke/{functionName} requests
func (g *Gateway) invokeHandler(w http.ResponseWriter, r *http.Request) {
    start := time.Now()

    // Extract function name from path
    functionName := extractFunctionName(r.URL.Path)
    if functionName == "" {
        http.Error(w, "Function name required", http.StatusBadRequest)
        return
    }

    // Parse invocation parameters
    req, err := g.parseInvocationRequest(r, functionName)
    if err != nil {
        http.Error(w, err.Error(), http.StatusBadRequest)
        return
    }

    // Route the request
```

```
    resp, err := g.router.Route(r.Context(), req)

    // Record metrics
    status := "200"
    if err != nil {
        status = errorToStatusCode(err)
    }
    g.metrics.requestsTotal.WithLabelValues(
        functionName,
        r.Method,
        status,
    ).Inc()
    g.metrics.requestDuration.WithLabelValues(
        functionName,
        r.Method,
    ).Observe(time.Since(start).Seconds())

    // Write response
    if err != nil {
        g.writeError(w, err)
        return
    }

    w.Header().Set("Content-Type", "application/octet-stream")
    w.Header().Set("X-Request-ID", resp.RequestID)
    w.Header().Set("X-Instance-ID", resp.InstanceID)
    w.WriteHeader(http.StatusOK)
    w.Write(resp.Payload)
}

// Helper functions (implemented elsewhere)
func extractFunctionName(path string) string { /* implementation */ }
func (g *Gateway) parseInvocationRequest(r *http.Request, fn string) (*types.InvocationRequest, error) { /* implementation */ }
func errorToStatusCode(err error) string { /* implementation */ }
func (g *Gateway) writeError(w http.ResponseWriter, err error) { /* implementation */ }
```

Core Logic Skeleton Code

Router with Least-Connections Load Balancing:

```go
// internal/gateway/router.go

package gateway

import (
    "context"
    "fmt"
    "sync"
    "sync/atomic"
    "time"

    "github.com/yourproject/internal/types"
)

// Router manages routing of invocation requests to function instances
type Router struct {
    mu              sync.RWMutex
    functions       map[string]*functionRoutingTable
    instanceManager InstanceManager // Interface to manage instances
    config          RouterConfig
}

// functionRoutingTable tracks instances and routing state for a single function
type functionRoutingTable struct {
    functionName     string
    instances        []*routedInstance
    concurrencyLimit int32
    activeCount      int32 // Atomic counter
    circuitBreaker   *circuit.Breaker
    lastScaleUpTime  time.Time
}

// routedInstance wraps a FunctionInstance with routing metadata
type routedInstance struct {
    instance      *types.FunctionInstance
    activeReqs    int32 // Atomic counter for least-connections
    lastUsed      time.Time
    failures      int32 // Recent failure count for circuit breaking
    healthy       bool
}

// RouterConfig holds routing configuration
type RouterConfig struct {
    DefaultConcurrencyLimit int
```

```
    CircuitBreakerThreshold int
    CircuitBreakerTimeout   time.Duration
    StickySessionTTL        time.Duration
}

// NewRouter creates a new router
func NewRouter(im InstanceManager, config RouterConfig) *Router {
    return &Router{
        functions:       make(map[string]*functionRoutingTable),
        instanceManager: im,
        config:          config,
    }
}

// Route routes an invocation request to an appropriate instance
func (r *Router) Route(ctx context.Context, req *types.InvocationRequest) (*types.InvocationResponse, error) {
    // TODO 1: Validate request is not expired (check req.IsExpired())
    // TODO 2: Lookup or create routing table for this function (use getOrCreateRoutingTable)
    // TODO 3: Check if function is at concurrency limit (compare activeCount vs limit)
    // TODO 4: If at limit and request is synchronous, attempt to queue (via queueManager)
    // TODO 5: Select instance using least-connections strategy (call selectInstance)
    // TODO 6: If no instance available but can scale, trigger scale-up and queue request
    // TODO 7: Forward request to selected instance (call forwardRequest)
    // TODO 8: On success, update instance metrics and return response
    // TODO 9: On failure, mark instance unhealthy, update circuit breaker, and retry if appropriate
    // TODO 10: Ensure atomic updates to activeCount and instance activeReqs counters
    // Hint: Use context.WithTimeout to ensure forwarding doesn't exceed request deadline
    // Hint: Implement retry logic only for idempotent operations (check request metadata)
    return nil, fmt.Errorf("not implemented")
}

// selectInstance chooses an instance using least-connections strategy
func (r *Router) selectInstance(rt *functionRoutingTable, req *types.InvocationRequest) (*routedInstance, error) {
    r.mu.RLock()
    defer r.mu.RUnlock()

    // TODO 1: Filter instances to only healthy, active ones (InstanceStateWarm or InstanceStateActive)
    // TODO 2: If req.PreferredInstanceID is set, try to use that instance if available and healthy
    // TODO 3: Among remaining instances, find the one with lowest activeReqs count
    // TODO 4: Handle tie-breakers (e.g., oldest instance, most recently used)
    // TODO 5: If no instances available, return appropriate error (ErrNoInstancesAvailable)
    // TODO 6: Increment the selected instance's activeReqs count atomically
```

```
    // Hint: Use atomic.LoadInt32 to read counters without full lock

    // Hint: Consider weighted least-connections if instances have different capacities

    return nil, fmt.Errorf("not implemented")

}


// forwardRequest sends the request to a specific instance

func (r *Router) forwardRequest(ctx context.Context, instance *routedInstance, req *types.InvocationRequest) (*types.InvocationResponse,
error) {

    // TODO 1: Build HTTP/gRPC request to instance endpoint (instance.instance.Host:Port)

    // TODO 2: Copy original request headers and body

    // TODO 3: Propagate timeout: calculate remaining time from ctx.Deadline()

    // TODO 4: Send request and await response

    // TODO 5: Parse response into InvocationResponse struct

    // TODO 6: Handle transport errors (network issues) vs. function errors (returned in response)

    // TODO 7: Update instance.lastUsed timestamp on success

    // TODO 8: On timeout, cancel the request and mark instance for health check

    // Hint: Use http.Client with timeout or context-aware transport

    // Hint: Consider connection pooling to instances for performance

    return nil, fmt.Errorf("not implemented")

}


// updateInstanceHealth updates circuit breaker based on success/failure

func (r *Router) updateInstanceHealth(instance *routedInstance, success bool) {

    // TODO 1: If success, reset failure count and mark healthy

    // TODO 2: If failure, increment failure count atomically

    // TODO 3: If failure count exceeds threshold, mark instance unhealthy

    // TODO 4: Schedule health check for unhealthy instances after timeout

    // TODO 5: Notify instance manager to potentially terminate and replace instance

    // Hint: Use exponential backoff for health check retries

}
```

Bounded Priority Request Queue:

```go
// internal/gateway/queue/queue.go

package queue

import (
    "container/heap"
    "context"
    "sync"
    "time"

    "github.com/yourproject/internal/types"
)

// QueuedRequest represents a request waiting for an instance
type QueuedRequest struct {
    Request    *types.InvocationRequest
    EnqueuedAt time.Time
    Priority   int // Lower number = higher priority
    index      int // Required by heap.Interface
}

// RequestQueue implements a bounded priority queue
type RequestQueue struct {
    mu       sync.Mutex
    heap     requestHeap
    maxSize  int
    notEmpty *sync.Cond
    notFull  *sync.Cond
    closed   bool
}

// NewRequestQueue creates a new bounded priority queue
func NewRequestQueue(maxSize int) *RequestQueue {
    q := &RequestQueue{
        heap:    make(requestHeap, 0),
        maxSize: maxSize,
    }
    q.notEmpty = sync.NewCond(&q.mu)
    q.notFull = sync.NewCond(&q.mu)
    heap.Init(&q.heap)
    return q
}

// Enqueue adds a request to the queue with timeout
```

```go
func (q *RequestQueue) Enqueue(ctx context.Context, req *QueuedRequest) error {
    q.mu.Lock()

    // TODO 1: Wait for queue not full with context timeout (use waitWithTimeout helper)
    // TODO 2: If queue is closed, return ErrQueueClosed
    // TODO 3: Set req.EnqueuedAt = time.Now()
    // TODO 4: Push request onto heap (heap.Push)
    // TODO 5: Signal notEmpty condition variable
    // TODO 6: Update queue depth metrics
    // Hint: Calculate priority based on req.Request.TimeRemaining() - shorter remaining time = higher priority

    q.mu.Unlock()
    return nil
}

// Dequeue removes and returns the highest priority request
func (q *RequestQueue) Dequeue(ctx context.Context) (*QueuedRequest, error) {
    q.mu.Lock()
    defer q.mu.Unlock()

    // TODO 1: Wait for queue not empty with context timeout
    // TODO 2: If queue is closed, return ErrQueueClosed
    // TODO 3: Pop highest priority request from heap (heap.Pop)
    // TODO 4: Signal notFull condition variable
    // TODO 5: Update queue depth metrics
    // TODO 6: Check if request has expired (req.Request.IsExpired()) - if so, discard and retry
    // Hint: Implement loop that skips expired requests until finding valid one or queue empty

    return nil, nil
}

// RemoveExpired removes and returns count of expired requests
func (q *RequestQueue) RemoveExpired(deadline time.Time) int {
    // TODO 1: Iterate through heap (without removing)
    // TODO 2: Find requests where req.Request.Deadline.Before(deadline)
    // TODO 3: Remove them from heap (maintaining heap property)
    // TODO 4: Return count of removed requests
    // TODO 5: Update metrics for expired requests
    // Hint: Build list of indices to remove, then rebuild heap or use heap.Remove

    return 0
}

// requestHeap implements heap.Interface for priority queue
type requestHeap []*QueuedRequest
```

```
func (h requestHeap) Len() int           { return len(h) }
func (h requestHeap) Less(i, j int) bool { return h[i].Priority < h[j].Priority }
func (h requestHeap) Swap(i, j int)      { h[i], h[j] = h[j], h[i]; h[i].index = i; h[j].index = j }

func (h *requestHeap) Push(x interface{}) {
    n := len(*h)
    item := x.(*QueuedRequest)
    item.index = n
    *h = append(*h, item)
}

func (h *requestHeap) Pop() interface{} {
    old := *h
    n := len(old)
    item := old[n-1]
    item.index = -1
    *h = old[0 : n-1]
    return item
}
```

Language-Specific Hints

Go-Specific Implementation Tips:

1. **Concurrency Control**: Use `sync.RWMutex` for routing table (many reads, few writes). Use `atomic` operations for instance request counters to avoid lock contention on hot paths.
2. **Context Propagation**: Always pass `context.Context` through function calls. Use `context.WithTimeout` to enforce deadlines. Check `ctx.Err()` before starting expensive operations.
3. **HTTP Client Pooling**: Create a shared `http.Client` with `Transport: &http.Transport{MaxIdleConnsPerHost: 100}` to reuse connections to function instances.
4. **Channel-based Queue Alternative**: For simpler implementations, use buffered channels as bounded queues: `make(chan *QueuedRequest, maxSize)`. Use `select` with `ctx.Done()` for timeouts.
5. **Prometheus Metrics**: Register metrics once, use `WithLabelValues` for dynamic labels. Consider using `promauto` for automatic registration.
6. **Graceful Shutdown**: Implement `Shutdown(ctx context.Context)` method that stops accepting new requests, waits for in-flight requests, and drains queues with context timeout.
7. **Connection Management**: Use `http.Client.Timeout` and `DialContext` for fine-grained control. Consider implementing connection health checks with `Transport.IdleConnTimeout`.

Milestone Checkpoint

After implementing request routing, verify functionality with these tests:

1. **Basic Routing Test**:

```bash
# Start gateway on port 8080
go run cmd/gateway/main.go --addr :8080


# Send test request
curl -X POST http://localhost:8080/invoke/myFunction \
  -H "Content-Type: application/json" \
  -d '{"test": "data"}' \
  -v
```

Expected: Gateway starts, accepts request, returns 503 (no instances available yet).

2. **Queue Test**:

```bash
# Configure function with max concurrency 1, queue size 5
# Start one instance, send 3 concurrent requests
ab -n 3 -c 3 -p test.json -T application/json \
  http://localhost:8080/invoke/myFunction
```

Expected: One request executes immediately, two queue. Check logs for queue depth metrics.

3. **Load Balancing Verification**:

```bash
# Start 3 instances, send 10 requests
for i in {1..10}; do
  curl -X POST http://localhost:8080/invoke/myFunction -d "req-$i" &
done
wait
```

Expected: Requests distributed across instances (check instance logs). Roughly equal distribution with least-connections.

4. **Timeout Test**:

```bash
# Set function timeout to 1s, send request that sleeps 2s
curl -X POST http://localhost:8080/invoke/slowFunction \
  -H "X-Function-Timeout: 1000" \
  -d '{"sleep": 2000}'
```

Expected: Request times out after 1s with 504 status or function error response.

Signs of Correct Implementation:

- Gateway starts without errors and binds to configured port
- Request logs show routing decisions (instance selection, queueing)
- Prometheus metrics show request counts, durations, queue depths
- Multiple instances receive balanced load
- Queue prevents overload when all instances busy
- Timeouts are respected and propagated

Common Issues and Diagnostics:

- **"Gateway returns 503 immediately"**: Check instance registration with router
- **"All requests go to one instance"**: Verify least-connections counters are updating
- **"Queue never drains"**: Check scale-up triggers and instance availability
- **"Memory grows unbounded"**: Verify queue bounds and request eviction
- **"Timeouts not working"**: Check deadline propagation through context chain

Component Design: Auto-Scaling

> **Milestone(s):** Milestone 5

The auto-scaling component is the **elastic nervous system** of the serverless runtime—it continuously monitors demand and dynamically adjusts the supply of execution environments to match workload patterns while respecting resource constraints and minimizing costs. This component embodies the core promise of serverless computing: scaling from zero to handle sudden bursts, then scaling back to zero when idle, all without human intervention. Unlike traditional autoscalers that operate on minute-long granularities, serverless auto-scaling must react within seconds to accommodate ephemeral, event-driven workloads while accounting for the latency penalty of **cold starts**.

Mental Model: The Elastic Concert Hall

Imagine managing a large concert hall with retractable seating sections. When ticket sales (requests) begin, you start with a minimal central section open. As the queue at the entrance (request queue) grows, you dynamically unfold additional seating sections (function instances) to accommodate more attendees. Each section takes time to deploy (cold start latency), so you monitor the arrival rate and sometimes pre-deploy sections based on historical patterns (predictive warming). When sections remain empty for a configured idle period, you retract them to save on heating and cleaning costs (scale-to-zero). The hall enforces fire code limits (maximum instances) and always keeps a few sections minimally prepared for VIP guests (provisioned concurrency). This elastic capacity management—balancing responsiveness against resource efficiency—captures the essence of auto-scaling in serverless systems.

Scaler Interface and Metrics

The auto-scaler operates as a control loop that periodically evaluates scaling decisions for each deployed function. It interacts with the **router** (to understand current request load and instance utilization), the **warm pool manager** (to provision or terminate instances), and the **metrics subsystem** (to gather historical data). The core interface abstracts these interactions:

Method Name	Parameters	Returns	Description
`EvaluateScaling(ctx context.Context, functionName string)`	`functionName`: Name of function to evaluate	`ScaleDecision` (struct containing direction and count)	Evaluates current metrics against scaling policies to determine if scaling action is needed
`RecordInvocationMetric(ctx context.Context, functionName string, timestamp time.Time)`	`functionName`: Function identifier, `timestamp`: When invocation occurred	`error`	Records an invocation event for rate calculation (called by gateway on each request)
`GetFunctionScalingConfig(ctx context.Context, functionName string)`	`functionName`: Function identifier	`*ScalingConfig`	Retrieves scaling configuration (min/max instances, thresholds, cooldowns)
`UpdateScalingConfig(ctx context.Context, functionName string, config ScalingConfig)`	`functionName`: Function identifier, `config`: New scaling configuration	`error`	Updates scaling configuration for a function
`StartScalingLoop(ctx context.Context)`	None	`error`	Starts the periodic scaling evaluation loop for all active functions
`StopScalingLoop(ctx context.Context)`	None	`error`	Stops the scaling evaluation loop

The auto-scaler's decisions are driven by two primary metrics computed over a sliding window (typically 15-60 seconds):

1. **Request Rate (RPS)**: Invocations per second, measured as a moving average. This indicates demand volume.
2. **Concurrent Executions**: Number of simultaneously active invocations across all instances of a function. This indicates utilization of existing capacity.

Additional derived metrics include:

- **Queue Depth**: Number of requests waiting in the per-function queue (from the router)
- **Instance Utilization**: `ConcurrentExecutions / (ActiveInstances * PerInstanceConcurrencyLimit)`
- **Cold Start Ratio**: Percentage of invocations experiencing cold starts

The core data structure for scaling configuration:

Field	Type	Description
`FunctionName`	`string`	Name of the function this configuration applies to
`MinInstances`	`int`	Minimum number of instances to keep running (including warm)
`MaxInstances`	`int`	Maximum number of instances allowed
`TargetConcurrency`	`int`	Desired number of concurrent executions per instance (used to calculate desired instance count)
`ScaleUpThreshold`	`float64`	Utilization threshold (0.0-1.0) that triggers scale-up (e.g., 0.8 = 80% utilization)
`ScaleDownThreshold`	`float64`	Utilization threshold (0.0-1.0) that triggers scale-down
`ScaleUpCooldown`	`time.Duration`	Minimum time between scale-up actions (prevents rapid oscillation)
`ScaleDownCooldown`	`time.Duration`	Minimum time between scale-down actions
`IdleTimeout`	`time.Duration`	How long an instance must be idle before being considered for termination (scale-to-zero)
`WarmPoolSize`	`int`	Number of instances to keep in warm pool (provisioned concurrency)
`MetricsWindow`	`time.Duration`	Time window for calculating metrics (e.g., 30s)

ADR: Scaling Algorithm - Reactive vs. Predictive

> **Decision: Threshold-Based Reactive Scaling with Predictive Warming Hints**
>
> - **Context**: The serverless runtime must scale function instances efficiently under unpredictable, bursty workloads typical of event-driven architectures. We need to choose between simple reactive approaches (responding to current load) and more complex predictive approaches (forecasting future load).
> - **Options Considered**:
>   1. **Pure Reactive Scaling**: Monitor current metrics (concurrency, queue depth) and scale when thresholds are breached.
>   2. **Predictive Scaling**: Use time-series forecasting (ARIMA, ML models) to predict future demand and pre-scale.
>   3. **Hybrid Approach**: Reactive scaling as baseline, with simple predictive warming for known patterns.
> - **Decision**: Implement threshold-based reactive scaling with optional predictive warming hints based on time-of-day/day-of-week patterns.
> - **Rationale**: Pure predictive scaling requires extensive historical data, adds significant complexity, and can be inaccurate for truly sporadic workloads. Reactive scaling is simpler, more robust, and sufficient for most serverless patterns. The hybrid approach gives us the best of both: reliability of reactive scaling with the ability to pre-warm for predictable patterns (like daily cron jobs or business-hour traffic). We avoid full ML-based prediction due to implementation complexity and the "cold start problem" for prediction models themselves.
> - **Consequences**:
>   - Positive: Simpler implementation, easier debugging, robust to unpredictable traffic.
>   - Negative: May miss optimization opportunities for highly predictable workloads.
>   - Trade-off: Accept slightly higher cold start latency for truly unpredictable bursts in exchange for system simplicity.

Option	Pros	Cons	Chosen?
Pure Reactive Scaling	Simple to implement, robust, no historical data needed, immediate response to actual load	Always reacts after load arrives (never pre-warms), can cause latency spikes during sudden bursts	Partially (as baseline)
Predictive Scaling	Can pre-warm before load arrives, minimizes cold starts for predictable patterns	Complex implementation, requires historical data, prediction inaccuracies can waste resources, "cold start" for the predictor itself	No
Hybrid Approach	Balances simplicity with optimization, can handle both unpredictable and predictable workloads	More complex than pure reactive, still requires some historical data for predictive part	**Yes**

The scaling algorithm operates as a periodic control loop with the following decision logic (illustrated in the flowchart `![Flowchart: Scale-Up Decision Logic](/api/project/serverless-runtime/architecture-doc/asset?path=diagrams%2Fflowchart-scaling-logic.svg)`):

1. **For each active function, every evaluation interval (e.g., 10 seconds):**
   1. Retrieve current metrics: `activeInstances`, `concurrentExecutions`, `queueDepth`
   2. Calculate desired instances: `desired = ceil(concurrentExecutions / targetConcurrency)`
   3. Apply bounds: `desired = max(minInstances, min(maxInstances, desired))`
   4. If `desired > activeInstances` AND `scaleUpCooldown` elapsed since last scale-up:
      - Trigger scale-up of `(desired - activeInstances)` instances
      - Record scale-up timestamp
   5. If `desired < activeInstances` AND `scaleDownCooldown` elapsed:

- Identify idle instances (those with `IdleDuration() > idleTimeout`)
- Mark up to `(activeInstances - desired)` idle instances as `InstanceStateDraining`

6. For scale-to-zero: If `activeInstances > minInstances` AND all instances have been idle beyond `idleTimeout`:
   - Mark all excess instances as `InstanceStateDraining`

The state transitions for scaling actions align with the instance lifecycle shown in `![Function Instance Lifecycle](/api/project/serverless-runtime/architecture-doc/asset?path=diagrams%2Finstance-state-machine.svg)`:

Current State	Event	Next State	Action Taken
`InstanceStateWarm`	Scale-down signal (instance selected for removal)	`InstanceStateDraining`	Instance stops accepting new requests, completes in-flight requests
`InstanceStateDraining`	All in-flight requests completed	`InstanceStateTerminated`	Container stopped and removed, resources freed
(No instance exists)	Scale-up signal (need new instance)	`InstanceStateProvisioning`	`CreateContainer` called, instance added to warm pool

Common Pitfalls in Auto-Scaling

⚠️ Pitfall: Scaling Thrashing (Rapid Oscillation)

- **Description**: The scaler repeatedly adds and removes instances in quick succession due to metrics fluctuating around thresholds.
- **Why it's wrong**: Causes instance churn, increased cold starts, wasted resources on container lifecycle operations, and potential request failures during transitions.
- **How to avoid**: Implement cooldown periods (`ScaleUpCooldown`, `ScaleDownCooldown`) that prevent another scaling action of the same direction for a minimum time. Use hysteresis by having different thresholds for scale-up (e.g., 80% utilization) and scale-down (e.g., 30% utilization).

⚠️ Pitfall: Ignoring Cold Start Latency in Scaling Decisions

- **Description**: Scaling logic treats newly provisioned instances as immediately available, but they incur cold start latency (seconds for some runtimes).
- **Why it's wrong**: During rapid traffic increase, newly scaled instances won't be ready in time to handle the surge, causing queue buildup and timeouts.
- **How to avoid**: Incorporate cold start latency into scaling calculations: when scaling up, assume new instances will be unavailable for their runtime's typical cold start duration. Alternatively, use predictive warming or maintain a small warm buffer.

⚠️ Pitfall: Over-Provisioning from Minimum Instance Counts

- **Description**: Setting `MinInstances > 0` for many functions "just to be safe," leading to constantly running idle instances.
- **Why it's wrong**: Violates the serverless cost model—you pay for idle capacity. For sporadic workloads, this can multiply costs significantly.
- **How to avoid**: Default `MinInstances` to 0. Use `WarmPoolSize` (provisioned concurrency) only for latency-critical functions with known baseline load. Educate users on the cost-performance trade-off.

⚠️ Pitfall: Not Accounting for Burst Traffic Patterns

- **Description**: The scaler uses smooth averages (e.g., 30-second RPS) and misses short, intense bursts that arrive and complete within the metrics window.
- **Why it's wrong**: Bursts cause timeouts or dropped requests because scaling doesn't react quickly enough.
- **How to avoid**: Include queue depth as a scaling signal—if queue is growing despite high utilization, scale up immediately. Use shorter metrics windows (e.g., 5 seconds) for highly variable workloads.

⚠️ Pitfall: Scale-Down Too Aggressive for Stateful Functions

- **Description**: Scaling down instances while they hold in-memory state (caches, WebSocket connections) not meant to be shared across instances.
- **Why it's wrong**: Terminating instances loses state, breaking application logic. Serverless functions should ideally be stateless, but some frameworks encourage in-memory caching for performance.
- **How to avoid**: Provide configuration to disable scale-to-zero for stateful functions, or implement graceful shutdown hooks that persist state to external storage before termination.

Implementation Guidance for Auto-Scaling

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Metrics Collection	In-memory counters with sliding window (Go `container/ring`)	Prometheus metrics with custom exporters
Scaling Loop	Periodic goroutine with `time.Ticker`	Kubernetes-style controller pattern with work queue
Configuration Storage	In-memory map (for prototype)	Persistent key-value store (etcd, Redis)
Predictive Warming	Time-of-day/day-of-week pattern matching	ML forecasting (Facebook Prophet, ARIMA)

B. Recommended File/Module Structure

```
project-root/
  cmd/scaler/main.go               # Scaler service entry point
  internal/scaler/
    scaler.go                      # Main scaling logic and control loop
    config.go                      # ScalingConfig type and validation
    metrics.go                     # Metrics collection and calculations
    predictor.go                   # Simple predictive warming (time-based patterns)
    scaler_test.go                 # Unit tests
  internal/gateway/                # (Existing) - calls RecordInvocationMetric
  internal/router/                 # (Existing) - provides queue depth and instance counts
  internal/pool/                   # (Existing) - WarmPoolManager for instance lifecycle
```

C. Infrastructure Starter Code (Metrics Collector)

```go
// internal/scaler/metrics.go

package scaler


import (

    "container/ring"

    "sync"

    "time"

)


// InvocationEvent records a single function invocation for rate calculation

type InvocationEvent struct {

    Timestamp time.Time

    FunctionName string

}


// MetricsCollector maintains sliding window metrics for scaling decisions

type MetricsCollector struct {

    mu sync.RWMutex

    // Per-function ring buffers of invocation timestamps

    invocationBuffers map[string]*ring.Ring

    windowSize time.Duration

    // Current concurrency counts

    concurrency map[string]int64

}


// NewMetricsCollector creates a collector with specified time window

func NewMetricsCollector(windowSize time.Duration) *MetricsCollector {

    return &MetricsCollector{

        invocationBuffers: make(map[string]*ring.Ring),

        windowSize:        windowSize,

        concurrency:       make(map[string]int64),

    }

}


// RecordInvocation adds an invocation timestamp for rate calculation

func (m *MetricsCollector) RecordInvocation(functionName string, ts time.Time) {

    m.mu.Lock()

    defer m.mu.Unlock()



    buf, exists := m.invocationBuffers[functionName]

    if !exists {

        // Estimate max invocations per window: assume 1000 RPS max
```

```
        maxEvents := int(m.windowSize.Seconds() * 1000)
        buf = ring.New(maxEvents)
        m.invocationBuffers[functionName] = buf
    }

    buf.Value = ts
    buf = buf.Next()
    m.invocationBuffers[functionName] = buf
}

// IncrementConcurrency increases concurrent execution count
func (m *MetricsCollector) IncrementConcurrency(functionName string) {
    m.mu.Lock()
    defer m.mu.Unlock()
    m.concurrency[functionName]++
}

// DecrementConcurrency decreases concurrent execution count
func (m *MetricsCollector) DecrementConcurrency(functionName string) {
    m.mu.Lock()
    defer m.mu.Unlock()
    m.concurrency[functionName]--
}

// GetRequestRate calculates requests per second over the sliding window
func (m *MetricsCollector) GetRequestRate(functionName string) float64 {
    m.mu.RLock()
    defer m.mu.RUnlock()

    buf, exists := m.invocationBuffers[functionName]
    if !exists {
        return 0.0
    }

    cutoff := time.Now().Add(-m.windowSize)
    count := 0

    // Walk the ring buffer counting events within window
    buf.Do(func(value interface{}) {
        if value == nil {
            return
        }
```

```
        ts := value.(time.Time)
        if ts.After(cutoff) {
            count++
        }
    })

    return float64(count) / m.windowSize.Seconds()
}

// GetConcurrency returns current concurrent executions
func (m *MetricsCollector) GetConcurrency(functionName string) int64 {
    m.mu.RLock()
    defer m.mu.RUnlock()
    return m.concurrency[functionName]
}
```

D. Core Logic Skeleton Code (Scaler Main Logic)

```go
// internal/scaler/scaler.go

package scaler


import (

    "context"

    "fmt"

    "log"

    "sync"

    "time"


    "github.com/yourproject/internal/pool"

    "github.com/yourproject/internal/router"

    "github.com/yourproject/internal/types"

)


// ScaleDecision represents a scaling action to take

type ScaleDecision struct {

    FunctionName string

    Direction    ScaleDirection // "up", "down", "none"

    Count        int           // Number of instances to add/remove

    Reason       string

}


type ScaleDirection string


const (

    ScaleUp   ScaleDirection = "up"

    ScaleDown ScaleDirection = "down"

    ScaleNone ScaleDirection = "none"

)


// Scaler is the main auto-scaling component

type Scaler struct {

    mu sync.RWMutex


    // Dependencies

    metricsCollector *MetricsCollector

    warmPool         pool.WarmPoolManager

    router           *router.Router

    configStore      ConfigStore


    // State
```

```
    activeFunctions   map[string]bool
    lastScaleUpTime   map[string]time.Time
    lastScaleDownTime map[string]time.Time

    // Control
    ticker      *time.Ticker
    stopChan    chan struct{}
    evalInterval time.Duration
}

// NewScaler creates a new auto-scaler
func NewScaler(mc *MetricsCollector, wp pool.WarmPoolManager,
               r *router.Router, interval time.Duration) *Scaler {
    return &Scaler{
        metricsCollector: mc,
        warmPool:         wp,
        router:           r,
        activeFunctions:  make(map[string]bool),
        lastScaleUpTime:  make(map[string]time.Time),
        lastScaleDownTime: make(map[string]time.Time),
        evalInterval:     interval,
        stopChan:         make(chan struct{}),
    }
}

// StartScalingLoop begins the periodic scaling evaluation
func (s *Scaler) StartScalingLoop(ctx context.Context) error {
    s.ticker = time.NewTicker(s.evalInterval)

    go func() {
        for {
            select {
            case <-s.ticker.C:
                s.evaluateAllFunctions(ctx)
            case <-s.stopChan:
                s.ticker.Stop()
                return
            case <-ctx.Done():
                s.ticker.Stop()
                return
            }
```

```
        }
    }()

    log.Printf("Scaling loop started with interval %v", s.evalInterval)
    return nil
}

// evaluateAllFunctions checks each active function for scaling needs
func (s *Scaler) evaluateAllFunctions(ctx context.Context) {
    s.mu.RLock()
    functions := make([]string, 0, len(s.activeFunctions))
    for fn := range s.activeFunctions {
        functions = append(functions, fn)
    }
    s.mu.RUnlock()

    for _, fn := range functions {
        decision := s.EvaluateScaling(ctx, fn)
        if decision.Direction != ScaleNone {
            s.executeScaling(ctx, decision)
        }
    }
}

// EvaluateScaling implements the core scaling algorithm
func (s *Scaler) EvaluateScaling(ctx context.Context, functionName string) ScaleDecision {
    // TODO 1: Retrieve scaling configuration for this function
    // config, err := s.configStore.GetFunctionScalingConfig(ctx, functionName)
    // If no config, return ScaleNone

    // TODO 2: Gather current metrics
    // requestRate := s.metricsCollector.GetRequestRate(functionName)
    // concurrency := s.metricsCollector.GetConcurrency(functionName)
    // queueDepth := s.router.GetQueueDepth(functionName)
    // activeInstances := s.router.GetActiveInstanceCount(functionName)

    // TODO 3: Calculate desired instances based on target concurrency
    // desired := int(math.Ceil(float64(concurrency) / float64(config.TargetConcurrency)))
    // Apply min/max bounds: desired = max(config.MinInstances, min(config.MaxInstances, desired))

    // TODO 4: Check cooldown periods
```

```
    // now := time.Now()
    // lastUp := s.lastScaleUpTime[functionName]
    // lastDown := s.lastScaleDownTime[functionName]

    // TODO 5: Determine scaling direction
    // if desired > activeInstances && now.Sub(lastUp) >= config.ScaleUpCooldown {
    //     return ScaleDecision{
    //         FunctionName: functionName,
    //         Direction:    ScaleUp,
    //         Count:        desired - activeInstances,
    //         Reason:       fmt.Sprintf("Concurrency %d exceeds threshold", concurrency),
    //     }
    // }

    // TODO 6: Check for scale-down (consider idle instances)
    // if desired < activeInstances && now.Sub(lastDown) >= config.ScaleDownCooldown {
    //     // Get idle instances from router
    //     idleCount := s.router.GetIdleInstanceCount(functionName, config.IdleTimeout)
    //     removeCount := min(activeInstances - desired, idleCount)
    //     if removeCount > 0 {
    //         return ScaleDecision{...}
    //     }
    // }

    // TODO 7: Check for scale-to-zero (if minInstances == 0 and all instances idle beyond timeout)

    return ScaleDecision{Direction: ScaleNone}
}

// executeScaling performs the actual scaling action
func (s *Scaler) executeScaling(ctx context.Context, decision ScaleDecision) {
    switch decision.Direction {
    case ScaleUp:
        // TODO 8: Trigger warm pool to create new instances
        // s.warmPool.PreWarm(ctx, decision.FunctionName, "latest", decision.Count)
        // Update lastScaleUpTime
        log.Printf("Scaling UP %s by %d instances: %s",
            decision.FunctionName, decision.Count, decision.Reason)

    case ScaleDown:
        // TODO 9: Get idle instances from router and mark them as DRAINING
```

```
        // The router will handle graceful termination after in-flight requests complete
        log.Printf("Scaling DOWN %s by %d instances: %s",
            decision.FunctionName, decision.Count, decision.Reason)
    }
}

// RecordInvocationMetric is called by gateway on each request
func (s *Scaler) RecordInvocationMetric(ctx context.Context,
                                        functionName string,
                                        timestamp time.Time) error {
    // TODO 10: Mark function as active for scaling consideration
    s.mu.Lock()
    s.activeFunctions[functionName] = true
    s.mu.Unlock()

    // Record the invocation for rate calculation
    s.metricsCollector.RecordInvocation(functionName, timestamp)
    return nil
}

// StopScalingLoop halts the scaling evaluation
func (s *Scaler) StopScalingLoop(ctx context.Context) error {
    close(s.stopChan)
    return nil
}
```

E. Language-Specific Hints

1. **Concurrency Safety**: Use `sync.RWMutex` for metrics maps that are read-heavy. The scaling loop reads metrics frequently while the gateway writes to them.
2. **Time Management**: Use `time.Ticker` for periodic evaluation rather than `time.Sleep` in a loop—it provides more consistent intervals.
3. **Context Propagation**: Pass `context.Context` through all scaling operations to allow cancellation during shutdown.
4. **Error Handling**: Scale operations should be best-effort. Log errors but don't fail the entire scaling loop for one function's failure.
5. **Memory Efficiency**: For the sliding window metrics, `container/ring` provides O(1) operations but requires type assertions. Consider a typed ring buffer for production.

F. Milestone Checkpoint

After implementing the auto-scaler, verify it works correctly:

1. **Start the system** with a sample Python function deployed:

```
go run cmd/server/main.go
```
BASH

2. **Send a burst of requests** using a load testing tool:

```
hey -n 100 -c 20 http://localhost:8080/invoke/my-function
```
BASH

3. **Monitor scaling behavior**:
   - Check logs for scale-up messages
   - Observe instance count increase in metrics (`GET /metrics` endpoint)
   - Wait for traffic to subside and verify scale-down occurs after cooldown period

- Verify scale-to-zero eventually terminates all instances (if `MinInstances=0`)

4. **Expected observations**:
   - Initial requests may experience cold starts
   - After scale-up, subsequent requests should use warm instances
   - Instance count should not exceed `MaxInstances`
   - After idle period, instances should terminate
5. **Signs of problems**:
   - No scaling occurs: Check metrics collection and threshold values
   - Constant scaling oscillations: Adjust cooldown periods or add hysteresis
   - Instances not terminating: Verify `IdleTimeout` and draining logic

Interactions and Data Flow

> **Milestone(s):** Milestone 4 (Request Routing) and Milestone 5 (Auto-Scaling), with dependencies on Milestone 2 (Execution Environment) and Milestone 3 (Cold Start Optimization)

This section traces the complete journey of a function invocation through the system, from the initial HTTP request to the final response. Understanding these interactions is critical to grasping how the architectural components collaborate to deliver the serverless experience. We'll examine both the **synchronous invocation flow** (where the client waits for the result) and the **lifecycle events** that govern function instance behavior.

Synchronous Invocation Flow

Think of this flow as a **restaurant dinner service**:

1. A customer (client) arrives and tells the host (Gateway) what they want to eat (which function to invoke)
2. The host checks if there's an available table (warm instance) with a free seat (concurrency capacity)
3. If not, the host might ask the kitchen to prepare a new table (cold start) or add the customer to a waitlist (queue)
4. Once seated, a waiter (Container) takes the order (request), the kitchen (function code) prepares the meal (executes), and the result is delivered back through the chain
5. After the meal, the table is cleaned (instance reset) and made available for the next customer

Complete Step-by-Step Walkthrough

The following numbered steps trace a synchronous HTTP function call from initial request to final response, highlighting the interactions between all system components:

1. **Client Request Submission**
   - The client sends an HTTP POST request to `https://gateway.example.com/invoke/{functionName}`
   - The request includes the function payload in the request body and any required headers
   - The client may specify a timeout via the `X-Timeout-Seconds` header or it defaults to the function's configured timeout
2. **Gateway Reception and Validation**
   - The `Gateway` receives the request through its HTTP server
   - It validates the function name exists in the system by checking with the `Router`
   - It enforces request size limits (configured via `GatewayConfig.MaxRequestBodyBytes`)
   - It creates an `InvocationRequest` with:
     - A unique `RequestID` for tracking
     - The function name and preferred version
     - The payload bytes
     - `Synchronous: true`
     - A `Deadline` calculated from the current time plus timeout
   - The gateway records metrics via `gatewayMetrics.requestsTotal` and starts a timer for request duration
3. **Router Request Processing**
   - The `Router.Route()` method is called with the `InvocationRequest`
   - The router looks up the `functionRoutingTable` for the specified function
   - It checks if the function's circuit breaker is open (too many recent failures) → if so, returns error immediately
   - It calls `selectInstance()` to choose an appropriate `FunctionInstance`

4. **Instance Selection Process**
   - **Warm Path (Ideal)**: If warm instances exist in the routing table:
     - The router applies **least-connections** strategy, preferring instances with lowest `activeReqs`
     - It checks that the selected instance's `activeReqs < concurrencyLimit` (per-instance concurrency)
     - It verifies the instance is healthy (not marked as unhealthy in `routedInstance.healthy`)
     - It increments the instance's `activeReqs` counter
   - **Cold Path (No Warm Instance Available)**:
     - The router calls the `Scaler` to potentially scale up
     - The scaler's `EvaluateScaling()` determines if a new instance should be created
     - If scaling is triggered, the scaler interacts with the `WarmPoolManager` to create a new instance
     - Meanwhile, the request may be queued (see step 5) or fail fast depending on configuration
5. **Request Queue Management (When No Instance Available)**
   - If all warm instances are at capacity (`activeReqs >= concurrencyLimit`):
     - The router creates a `QueuedRequest` with priority based on request deadline (earlier deadline = higher priority)
     - It calls `RequestQueue.Enqueue()` to add the request to the function-specific queue
     - The gateway sets an HTTP timeout and waits for dequeue notification
     - The `gatewayMetrics.queueDepth` is incremented to monitor queue size
   - **Queue Full Scenario**: If the queue reaches its `maxSize`, new requests receive HTTP 503 "Service Unavailable"
6. **Instance Acquisition and Preparation**
   - Once an instance is selected (or becomes available):
     - The router calls `WarmPoolManager.AcquireWarmInstance()` for the function
     - The warm pool manager:
       - Marks the instance state as `InstanceStateActive` from `InstanceStateWarm`
       - Updates `LastUsedAt` timestamp
       - Returns the `FunctionInstance` with its container details (host, port)
     - The router records the instance assignment in the `InvocationRequest.PreferredInstanceID`
7. **Request Forwarding to Instance**
   - The router calls `forwardRequest()` which:
     - Establishes an HTTP connection to the instance's container (at `instance.Host:instance.Port`)
     - Forwards the original request payload with additional headers:
       - `X-Function-Request-ID`: The unique request identifier
       - `X-Deadline-Timestamp`: The absolute deadline as Unix timestamp
       - `X-Instance-ID`: The instance identifier for tracking
     - Sets appropriate timeouts based on the remaining deadline
   - The instance container (running the function runtime) receives and processes the request
8. **Function Execution Within Container**
   - The container's runtime (e.g., a Go HTTP server) receives the forwarded request
   - It extracts the function handler name from environment variables
   - It loads the function code (already present in the container from packaging)
   - It invokes the handler function with the request payload
   - The handler executes with the configured resource limits (monitored via cgroups)
   - The runtime captures stdout/stderr and any returned values
9. **Response Propagation Back Through Chain**
   - The instance container sends an HTTP response back to the router's `forwardRequest()`
   - The router:
     - Decrements the instance's `activeReqs` counter
     - Calls `updateInstanceHealth()` with success/failure based on HTTP status code
     - If successful, marks the instance as healthy; if failed, increments failure count
   - The router packages the response into an `InvocationResponse` with:
     - The response payload

    - Duration metrics
    - Memory usage (from container stats)
    - Any errors that occurred

10. **Instance Release and Cleanup**
    - The router calls `WarmPoolManager.ReleaseInstance()` with the instance and health status
    - The warm pool manager:
        - If the instance is healthy and under its TTL/idle timeout, returns it to the pool (state `InstanceStateWarm`)
        - If unhealthy or expired, triggers cleanup via `CleanupInstance()`
        - Updates `LastUsedAt` timestamp
    - The `Scaler` records the completion via `DecrementConcurrency()` for concurrency tracking

11. **Final Response to Client**
    - The gateway receives the `InvocationResponse` from the router
    - It sets appropriate HTTP status code (200 for success, 5xx for errors)
    - It returns the response payload in the HTTP body
    - It adds response headers:
        - `X-Request-ID`: For client correlation
        - `X-Function-Duration-MS`: Execution time in milliseconds
        - `X-Billed-Duration-MS`: Rounded-up billing duration
    - The gateway records final metrics:
        - Request duration in `gatewayMetrics.requestDuration`
        - Success/error counts in `gatewayMetrics.errorsTotal`

Cold Start vs Warm Start Variation

The key difference between cold and warm starts occurs in steps 4-6:

Warm Start Path (Sub-100ms):

```
Router → WarmPoolManager.AcquireWarmInstance() → existing warm container → execute function
```

- No container creation overhead
- Runtime already initialized
- Dependencies pre-loaded
- Typically completes in <100ms total

Cold Start Path (500ms-2000ms):

```
Router → Scaler → WarmPoolManager.PreWarm() → ContainerManager.CreateContainer() →
ContainerManager.StartContainer() → dependency loading → execute function
```

- Full container creation (100-500ms)
- Image pulling if not cached (variable, 0-2000ms)
- Runtime initialization (JVM: 500ms+, Go: <50ms)
- Dependency loading/imports (variable)
- Typically 500ms-2000ms total

> **Key Insight:** The system's performance goal is to maximize warm starts through intelligent pooling and predictive warming while maintaining the flexibility to handle unpredictable loads with cold starts when necessary.

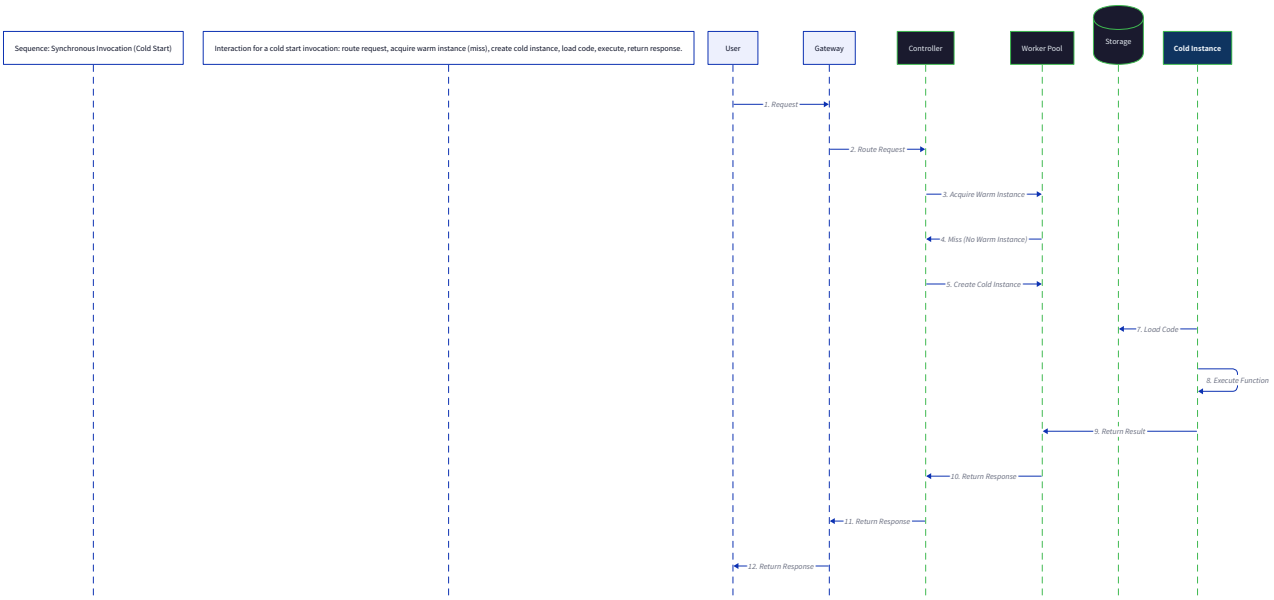


Diagram showing the complete interaction sequence between User, Gateway, Router, Scaler, WarmPoolManager, ContainerManager, and Storage for a cold start invocation.

Data Transformation Through the Pipeline

Stage	Input Data	Output Data	Key Transformation
Client → Gateway	Raw HTTP request	`InvocationRequest`	HTTP headers/body packaged into structured request
Gateway → Router	`InvocationRequest`	`QueuedRequest` (if queued)	Priority calculation based on deadline
Router → WarmPool	Function name, version	`FunctionInstance`	Instance selection from pool or creation
Router → Container	`InvocationRequest`	HTTP request to container	Request forwarding with metadata headers
Container → Function	HTTP request	Handler invocation	Payload deserialization to function parameters
Function → Container	Return value	HTTP response	Result serialization to HTTP response
Container → Router	HTTP response	`InvocationResponse`	Response packaging with metrics
Router → Gateway	`InvocationResponse`	HTTP response	Structured response to HTTP translation
Gateway → Client	HTTP response	Raw HTTP response	Final delivery to client

Function Instance Lifecycle Events

Think of a function instance's lifecycle as a **theater actor's backstage workflow**:

- **PROVISIONING**: Actor getting into costume and makeup (container being created)
- **WARM**: Actor waiting in the wings, ready to go on stage (instance idle in pool)
- **ACTIVE**: Actor performing on stage (executing a request)
- **DRAINING**: Actor taking final bow but still on stage (finishing requests during scale-down)
- **TERMINATED**: Actor has left the theater (container destroyed)
- **ERROR**: Actor fell ill and can't perform (container in error state)

Each state transition is triggered by specific events in the system, which we'll explore through both a state machine table and detailed event descriptions.

Instance State Transition Table

The following table defines all possible state transitions for a `FunctionInstance`, including the triggering events, conditions, and resulting actions:

Current State	Event	Next State	Conditions	Actions Taken
(none)	`CreateInstance`	`InstanceStateProvisioning`	Valid function definition exists	1. Call `ContainerManager.CreateContai` 2. Set `CreatedAt` timestamp 3. Generate unique `ID`
`InstanceStateProvisioning`	`ContainerStarted`	`InstanceStateWarm`	Container started successfully	1. Mark container as running 2. Initialize runtime environment 3. Pre-load dependencies 4. Add to warm pool
`InstanceStateProvisioning`	`ContainerStartFailed`	`InstanceStateError`	Container failed to start	1. Record error details 2. Schedule cleanup 3. Trigger scaling retry if needed
`InstanceStateWarm`	`RequestAssigned`	`InstanceStateActive`	Instance selected by router	1. Increment `InvocationCount` 2. Update `LastUsedAt` 3. Mark as active in routing table
`InstanceStateActive`	`RequestCompleted`	`InstanceStateWarm`	Request finished successfully AND instance under TTL/idle limits	1. Decrement active request count 2. Reset container state if needed 3. Return to warm pool
`InstanceStateActive`	`RequestFailed`	`InstanceStateWarm` or `InstanceStateError`	Request failed If recoverable error → WARM If unrecoverable → ERROR	Recoverable: Reset container Unrecoverable: Mark unhealthy, schedule replacement
`InstanceStateWarm`	`TTLExpired`	`InstanceStateDraining`	Instance age > `InstanceTTL`	1. Mark for termination 2. Remove from warm pool 3. Start graceful shutdown
`InstanceStateWarm`	`IdleTimeout`	`InstanceStateDraining`	`IdleDuration()` > `IdleTimeout`	1. Mark for termination 2. Remove from routing table 3. Start graceful shutdown
`InstanceStateWarm`	`ScaleDownSignal`	`InstanceStateDraining`	Scaler determines excess capacity	1. Select instance for termination (oldest/ used) 2. Begin drain process
`InstanceStateActive`	`ScaleDownSignal`	`InstanceStateDraining`	Scaler needs to reduce capacity AND instance has no active requests	1. Wait for current request to complete 2. Transition to DRAINING after completi
`InstanceStateDraining`	`DrainCompleted`	`InstanceStateTerminated`	All requests completed OR force timeout reached	1. Call `ContainerManager.StopContainer` 2. Call `ContainerManager.RemoveContai` 3. Update `TerminatedAt` timestamp
`InstanceStateDraining`	`ForceTerminate`	`InstanceStateTerminated`	Scale-down timeout reached OR system shutdown	1. Force kill container processes 2. Remove container resources 3. Clean up network/volumes
`InstanceStateError`	`HealthCheckFailed`	`InstanceStateTerminated`	Container unrecoverable OR consecutive failures	1. Log error details 2. Force remove container 3. Notify monitoring system
Any state	`SystemShutdown`	`InstanceStateTerminated`	System graceful shutdown initiated	1. Stop accepting new requests 2. Wait for in-flight requests 3. Terminate all containers

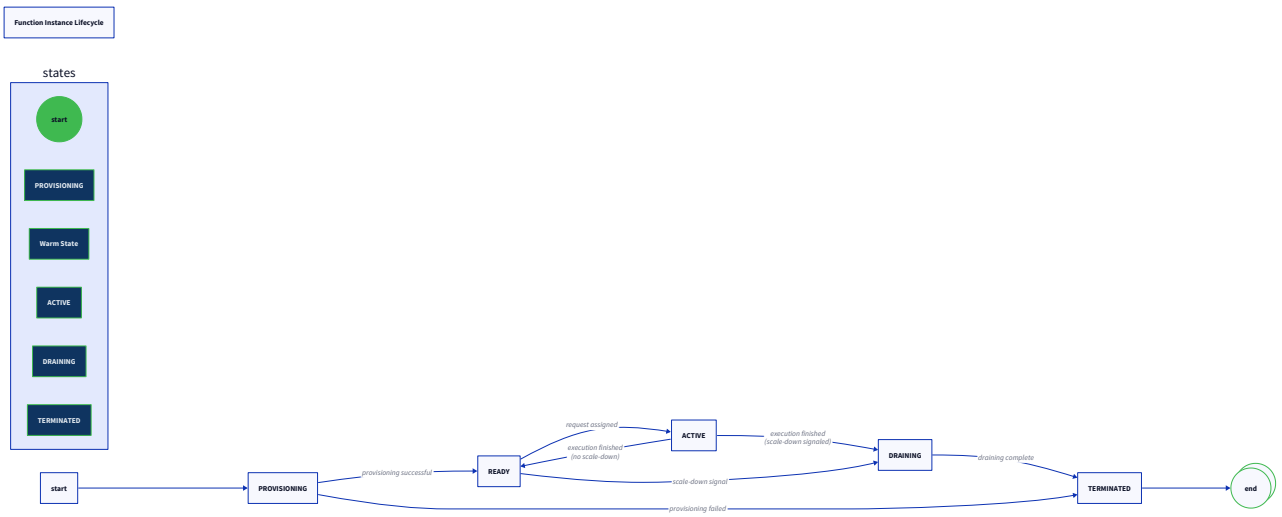


Diagram showing the state machine for a FunctionInstance with transitions between PROVISIONING, WARM, ACTIVE, DRAINING, TERMINATED, and ERROR states.

Detailed Event Descriptions

Each event in the state machine corresponds to specific system operations or conditions:

1. CreateInstance Event

- **Triggered by**: `WarmPoolManager.PreWarm()` or `Scaler.executeScaling(ScaleUp)`
- **Source**: Auto-scaling logic or predictive warming
- **Data**: `FunctionDefinition` containing runtime, handler, resource limits
- **Outcome**: Container creation initiated via `ContainerManager`

2. ContainerStarted Event

- **Triggered by**: Successful return from `ContainerManager.StartContainer()`
- **Source**: Container management layer
- **Data**: Container ID, host/port assignment, startup duration
- **Outcome**: Instance marked as ready, added to routing table and warm pool

3. RequestAssigned Event

- **Triggered by**: `Router.selectInstance()` choosing this instance
- **Source**: Request routing layer
- **Data**: `InvocationRequest` details, request deadline
- **Outcome**: Instance state changes to ACTIVE, concurrency counters incremented

4. RequestCompleted Event

- **Triggered by**: Successful return from `forwardRequest()`
- **Source**: Request routing layer after instance responds
- **Data**: `InvocationResponse` with duration, memory usage, result
- **Outcome**: Instance returned to pool if healthy, metrics recorded

5. RequestFailed Event

- **Triggered by**: Error from `forwardRequest()` (timeout, crash, error response)
- **Source**: Request routing layer or container health check
- **Data**: Error type, stack trace (if available), HTTP status code
- **Outcome**: Instance may be retired (ERROR) or reset (WARM) based on error severity

6. TTLExpired Event

- **Triggered by**: `WarmPoolManager.cleanupExpired()` periodic check
- **Source**: Warm pool maintenance goroutine
- **Condition**: `instance.Age() > PoolConfig.InstanceTTL`

- **Outcome**: Instance removed from pool, scheduled for termination

7. IdleTimeout Event

- **Triggered by**: `WarmPoolManager.cleanupExpired()` periodic check
- **Source**: Warm pool maintenance goroutine
- **Condition**: `instance.IdleDuration() > PoolConfig.IdleTimeout`
- **Outcome**: Instance marked for scale-to-zero termination

8. ScaleDownSignal Event

- **Triggered by**: `Scaler.EvaluateScaling()` returning `ScaleDown` decision
- **Source**: Auto-scaling component
- **Condition**: Low utilization metrics and cooldown period elapsed
- **Outcome**: Instance transitioned to DRAINING state, removed from routing

9. DrainCompleted Event

- **Triggered by**: All active requests finished on DRAINING instance
- **Source**: Router monitoring request completion
- **Condition**: `instance.activeReqs == 0` for DRAINING instance
- **Outcome**: Container termination process initiated

10. ForceTerminate Event

- **Triggered by**: Drain timeout expiration or system shutdown
- **Source**: Drain timeout timer or shutdown coordinator
- **Condition**: DRAINING state duration > max drain timeout
- **Outcome**: Forced container kill, immediate resource cleanup

11. HealthCheckFailed Event

- **Triggered by**: Container health check failure or OOM kill
- **Source**: Container manager health monitoring
- **Condition**: Container process died or unreachable
- **Outcome**: Emergency cleanup, error logging, replacement instance creation

Event-Driven Interactions Between Components

The following table shows how different system components interact through these lifecycle events:

Event	Primary Component	Secondary Component	Communication Method
`CreateInstance`	`Scaler` →	`WarmPoolManager`	Direct method call
`ContainerStarted`	`ContainerManager` →	`WarmPoolManager`	Callback/notification channel
`RequestAssigned`	`Router` →	`FunctionInstance`	Direct state mutation
`RequestCompleted`	`Router` →	`WarmPoolManager`	`ReleaseInstance()` call
`ScaleDownSignal`	`Scaler` →	`Router`	Update routing table
`DrainCompleted`	`Router` →	`ContainerManager`	`StopContainer()` call
`HealthCheckFailed`	Health monitor →	`WarmPoolManager`	Error channel notification

Lifecycle Metrics and Monitoring

Each state transition generates valuable metrics for system observability:

Metric	Calculation	Purpose
Provisioning Duration	`ContainerStarted` timestamp - `CreateInstance` timestamp	Measure cold start container creation time
Warm Pool Size	Count of instances in `InstanceStateWarm`	Monitor resource allocation for warm starts
Active Invocations	Count of instances in `InstanceStateActive`	Measure current system load
Drain Time	`DrainCompleted` timestamp - ScaleDownSignal timestamp	Assess graceful shutdown efficiency
Instance Lifetime	`TerminatedAt` timestamp - `CreatedAt` timestamp	Calculate container churn rate
Error Rate	Instances transitioning to `InstanceStateError` / Total instances	Measure system stability

Concurrency and State Consistency

Maintaining consistent state across concurrent operations requires careful coordination:

1. **State Transition Locking**: Each `FunctionInstance` has a mutex protecting state field updates
2. **Atomic Counters**: `activeReqs` uses `int32` with atomic operations for concurrency safety
3. **Optimistic Concurrency Control**: Before transitioning from WARM to ACTIVE, verify `activeReqs < concurrencyLimit` with atomic check
4. **Event Serialization**: Lifecycle events for the same instance are processed sequentially through the router's state machine

> **Critical Design Principle:** The instance lifecycle follows a **single-writer principle**—only the component that "owns" the current state (Router for ACTIVE, WarmPoolManager for WARM, Scaler for PROVISIONING) can initiate state transitions for that instance, preventing race conditions.

Recovery from Mid-Transition Failures

The system must handle cases where events fail mid-transition:

Failure Scenario	Detection Method	Recovery Action
Container creation timeout	`CreateContainer()` timeout	Mark as ERROR, retry with exponential backoff
State transition deadlock	Watchdog timer on state duration	Force transition based on heuristic (e.g., if PROVISIONING > 2min → ERROR)
Lost instance (network partition)	Health check failures + lack of heartbeats	Mark as ERROR, remove from routing tables, create replacement
Orphaned ACTIVE instance	No request completion after timeout + no router assignment	Force transition to ERROR via cleanup sweep

The interaction patterns and lifecycle events described here form the operational heartbeat of the serverless runtime. By understanding these flows, developers can effectively debug issues, optimize performance, and extend the system with new capabilities while maintaining the consistency and reliability expected from a production-grade serverless platform.

Implementation Guidance

Technology Recommendations Table

Component	Simple Option	Advanced Option
Request/Response Flow	HTTP/1.1 with Keep-Alive (`net/http`)	HTTP/2 with multiplexing (`golang.org/x/net/http2`)
Internal Communication	Direct method calls + channels	gRPC with streaming for instance updates
State Synchronization	`sync.RWMutex` per instance	Distributed consensus (etcd/Raft) for multi-node
Event Propagation	Go channels with select loops	Message bus (NATS, Kafka) for distributed events
Timeout Management	`context.WithTimeout` + deadlines	Hierarchical timeout chains with propagation
Queue Implementation	Heap-based priority queue (`container/heap`)	Redis-backed queue with persistence

Recommended File/Module Structure

```
project-root/
  cmd/
    gateway/                # HTTP gateway entry point
      main.go
    controller/             # Management plane entry point
      main.go
  internal/
    gateway/                # Request routing component
      gateway.go            # HTTP server and middleware
      router.go             # Request routing logic
      queue.go              # Request queue implementation
      loadbalancer.go       # Instance selection algorithms
      metrics.go            # Gateway-specific metrics
    lifecycle/              # Instance lifecycle management
      events.go             # Lifecycle event definitions
      statemachine.go       # State transition logic
      healthcheck.go        # Instance health monitoring
    flows/                  # Data flow orchestration
      synchronous.go        # Synchronous invocation flow
      asynchronous.go       # Async invocation flow (future)
      errorhandling.go      # Flow-level error recovery
    integration/            # Component integration points
      coordinator.go        # Orchestrates cross-component flows
      hooks.go              # Lifecycle hooks for extensibility
```

Infrastructure Starter Code

Complete Flow Coordinator (Ready to Use):

```go
// internal/integration/coordinator.go

package integration

import (
    "context"
    "time"
    "github.com/ourproject/internal/gateway"
    "github.com/ourproject/internal/lifecycle"
    "github.com/ourproject/internal/types"
)

// FlowCoordinator orchestrates the complete invocation flow
type FlowCoordinator struct {
    router        *gateway.Router
    warmPool      pool.WarmPoolManager
    scaler        *scaler.Scaler
    metrics       *types.Metrics
    eventBus      chan lifecycle.InstanceEvent
    timeoutConfig TimeoutConfig
}

type TimeoutConfig struct {
    GatewayTimeout     time.Duration
    RoutingTimeout     time.Duration
    ForwardingTimeout  time.Duration
    DrainTimeout       time.Duration
    HealthCheckTimeout time.Duration
}

// NewFlowCoordinator creates a new coordinator
func NewFlowCoordinator(router *gateway.Router, warmPool pool.WarmPoolManager,
    scaler *scaler.Scaler, metrics *types.Metrics) *FlowCoordinator {
    return &FlowCoordinator{
        router:   router,
        warmPool: warmPool,
        scaler:   scaler,
        metrics:  metrics,
        eventBus: make(chan lifecycle.InstanceEvent, 100),
        timeoutConfig: TimeoutConfig{
            GatewayTimeout:     30 * time.Second,
            RoutingTimeout:     5 * time.Second,
            ForwardingTimeout:  25 * time.Second,
```

```
            DrainTimeout:       10 * time.Second,
            HealthCheckTimeout: 2 * time.Second,
        },
    }
}

// StartEventProcessor processes lifecycle events
func (fc *FlowCoordinator) StartEventProcessor(ctx context.Context) {
    go func() {
        for {
            select {
            case <-ctx.Done():
                return
            case event := <-fc.eventBus:
                fc.handleEvent(ctx, event)
            }
        }
    }()
}

// handleEvent processes a single lifecycle event
func (fc *FlowCoordinator) handleEvent(ctx context.Context, event lifecycle.InstanceEvent) {
    switch event.Type {
    case lifecycle.EventContainerStarted:
        fc.onContainerStarted(ctx, event)
    case lifecycle.EventRequestCompleted:
        fc.onRequestCompleted(ctx, event)
    case lifecycle.EventHealthCheckFailed:
        fc.onHealthCheckFailed(ctx, event)
    }
}

// Helper for consistent timeout handling
func (fc *FlowCoordinator) withTimeout(ctx context.Context, timeout time.Duration,
    operation string) (context.Context, context.CancelFunc) {
    fc.metrics.OperationDuration.WithLabelValues(operation).Observe(0)
    start := time.Now()
    ctx, cancel := context.WithTimeout(ctx, timeout)

    // Record duration on completion
    go func() {
        <-ctx.Done()
```

```
        duration := time.Since(start).Seconds()
        fc.metrics.OperationDuration.WithLabelValues(operation).Observe(duration)
    }()

    return ctx, cancel
}
```

Lifecycle Event Definitions (Ready to Use):

```go
// internal/lifecycle/events.go

package lifecycle

import (
    "time"
    "github.com/ourproject/internal/types"
)

// InstanceEventType represents types of lifecycle events
type InstanceEventType string

const (
    EventCreateInstance      InstanceEventType = "create_instance"
    EventContainerStarted    InstanceEventType = "container_started"
    EventRequestAssigned     InstanceEventType = "request_assigned"
    EventRequestCompleted    InstanceEventType = "request_completed"
    EventRequestFailed       InstanceEventType = "request_failed"
    EventTTLExpired          InstanceEventType = "ttl_expired"
    EventIdleTimeout         InstanceEventType = "idle_timeout"
    EventScaleDownSignal     InstanceEventType = "scale_down_signal"
    EventDrainCompleted      InstanceEventType = "drain_completed"
    EventForceTerminate      InstanceEventType = "force_terminate"
    EventHealthCheckFailed   InstanceEventType = "health_check_failed"
)

// InstanceEvent represents a lifecycle event
type InstanceEvent struct {
    Type        InstanceEventType
    InstanceID  string
    FunctionName string
    Timestamp   time.Time
    Data        map[string]interface{}
    Source      string
}

// NewInstanceEvent creates a new lifecycle event
func NewInstanceEvent(eventType InstanceEventType, instance *types.FunctionInstance,
    source string) InstanceEvent {
    return InstanceEvent{
        Type:        eventType,
        InstanceID:  instance.ID,
        FunctionName: instance.FunctionName,
        Timestamp:   time.Now(),
```

```
        Data:        make(map[string]interface{}),
        Source:      source,
    }
}

// EventDispatcher manages event distribution
type EventDispatcher struct {
    subscribers map[string]chan InstanceEvent
    mu          sync.RWMutex
}

// Subscribe adds a subscriber for events
func (ed *EventDispatcher) Subscribe(id string, bufferSize int) <-chan InstanceEvent {
    ed.mu.Lock()
    defer ed.mu.Unlock()

    ch := make(chan InstanceEvent, bufferSize)
    ed.subscribers[id] = ch
    return ch
}

// Publish sends an event to all subscribers
func (ed *EventDispatcher) Publish(event InstanceEvent) {
    ed.mu.RLock()
    defer ed.mu.RUnlock()

    for _, ch := range ed.subscribers {
        select {
        case ch <- event:
            // Event sent successfully
        default:
            // Channel full, drop event (monitor this)
        }
    }
}
```

Core Logic Skeleton Code

Synchronous Invocation Handler (To Implement):

```go
// internal/flows/synchronous.go

package flows

import (
    "context"
    "net/http"
    "time"
    "github.com/ourproject/internal/gateway"
    "github.com/ourproject/internal/types"
    "github.com/ourproject/internal/lifecycle"
)

// SynchronousInvoker handles synchronous function invocations
type SynchronousInvoker struct {
    router       *gateway.Router
    eventDispatcher *lifecycle.EventDispatcher
    metrics      *types.Metrics
    config       SyncInvokerConfig
}

type SyncInvokerConfig struct {
    MaxQueueTime      time.Duration
    RetryOnColdStart  bool
    MaxRetries        int
    EnableStickyRouting bool
}

// Invoke handles a synchronous invocation request
func (si *SynchronousInvoker) Invoke(ctx context.Context,
    req *types.InvocationRequest) (*types.InvocationResponse, error) {

    // TODO 1: Validate the request has all required fields
    // Check req.FunctionName, req.Payload, req.Deadline
    // Return error if validation fails

    // TODO 2: Record invocation start in metrics
    // Use si.metrics.RecordInvocation() to track start time
    // Increment concurrency counter for the function

    // TODO 3: Route the request through the router
    // Call si.router.Route(ctx, req) to get response
    // Handle context cancellation and deadlines
```

```
    // TODO 4: Process the response
    // If error, determine if retry is appropriate (cold start timeout)
    // For retryable errors, implement exponential backoff with MaxRetries

    // TODO 5: Record completion metrics
    // Record duration, memory usage, success/failure
    // Decrement concurrency counter

    // TODO 6: Handle sticky session if enabled
    // If EnableStickyRouting and successful, store instance preference
    // For subsequent requests from same client, use PreferredInstanceID

    // TODO 7: Return final response or error
    // Ensure error wrapping for proper error types (timeout vs. function error)

    return nil, nil // Replace with actual implementation
}

// handleColdStartRetry determines if a cold start timeout should be retried
func (si *SynchronousInvoker) handleColdStartRetry(ctx context.Context,
    originalErr error, req *types.InvocationRequest, attempt int) (bool, error) {
    // TODO 1: Check if error is a cold start timeout
    // Look for "container startup timeout" or similar patterns

    // TODO 2: Verify we haven't exceeded MaxRetries
    // Return false if attempt >= si.config.MaxRetries

    // TODO 3: Check if retry is enabled in config
    // Return false if !si.config.RetryOnColdStart

    // TODO 4: Calculate backoff delay
    // Use exponential backoff: delay = baseDelay * (2^attempt)
    // Implement jitter to avoid thundering herd

    // TODO 5: Wait for backoff period with context awareness
    // Use time.Sleep or <-time.After with select on ctx.Done()

    // TODO 6: Return true to indicate retry should be attempted
    return false, nil // Replace with actual implementation
```

```
}
```

Instance State Machine (To Implement):

GO

```go
// internal/lifecycle/statemachine.go

package lifecycle


import (

    "context"

    "sync"

    "time"

    "github.com/ourproject/internal/types"

)


// StateMachine manages state transitions for FunctionInstance

type StateMachine struct {

    instance *types.FunctionInstance

    mu       sync.RWMutex

    handlers map[types.FunctionInstanceState]StateHandler

    eventCh  chan InstanceEvent

}


// StateHandler processes events for a specific state

type StateHandler interface {

    HandleEvent(ctx context.Context, event InstanceEvent,

        instance *types.FunctionInstance) (types.FunctionInstanceState, error)

}


// Transition performs a state transition with validation

func (sm *StateMachine) Transition(ctx context.Context,

    newState types.FunctionInstanceState, reason string) error {


    sm.mu.Lock()

    defer sm.mu.Unlock()


    // TODO 1: Validate transition is allowed

    // Check validTransitions[sm.instance.State][newState] map

    // Return error if transition is invalid


    // TODO 2: Execute exit actions for current state

    // Call sm.handlers[sm.instance.State].Exit() if exists

    // Clean up resources specific to current state


    // TODO 3: Update instance state

    // Set sm.instance.State = newState

    // Update timestamps based on state (LastUsedAt for ACTIVE, etc.)
```

```
    // TODO 4: Execute entry actions for new state
    // Call sm.handlers[newState].Enter() if exists
    // Initialize resources needed for new state

    // TODO 5: Log the transition for observability
    // Use structured logging with instance ID, old state, new state, reason

    // TODO 6: Publish state change event
    // Send to eventCh for subscribers (monitoring, scaling decisions)

    // TODO 7: Update metrics
    // Decrement counter for old state, increment for new state

    return nil // Replace with actual implementation
}

// ProcessEvent handles incoming lifecycle events
func (sm *StateMachine) ProcessEvent(ctx context.Context, event InstanceEvent) error {
    // TODO 1: Get current state handler
    // handler := sm.handlers[sm.instance.State]
    // Return error if no handler for current state

    // TODO 2: Delegate event handling to state-specific handler
    // newState, err := handler.HandleEvent(ctx, event, sm.instance)
    // Return error if handler returns error

    // TODO 3: Transition to new state if different
    // if newState != sm.instance.State {
    //     return sm.Transition(ctx, newState, string(event.Type))
    // }

    // TODO 4: Handle same-state events (e.g., heartbeat in WARM state)
    // Update instance metadata without state change

    return nil // Replace with actual implementation
}

// validateTransition defines allowed state transitions
func (sm *StateMachine) validateTransition(oldState, newState types.FunctionInstanceState) bool {
    // TODO: Implement validation logic based on state transition table
```

```
    // Return true only for allowed transitions
    // Example: PROVISIONING -> WARM (allowed), WARM -> PROVISIONING (not allowed)
    return false // Replace with actual implementation
}
```

Language-Specific Hints

1. **Context Propagation**: Use `context.Context` consistently through the call chain. Create derived contexts with `context.WithTimeout()` for each stage (gateway, routing, forwarding) to ensure proper timeout cascading.
2. **Error Wrapping**: Use `fmt.Errorf()` with `%w` verb to wrap errors while preserving the original error type for downstream handling decisions.
3. **Concurrent State Updates**: Use `sync.RWMutex` for `FunctionInstance` state fields. For counters like `activeReqs`, use `atomic` operations (`atomic.AddInt32`, `atomic.LoadInt32`) for better performance.
4. **Channel Patterns**: Use buffered channels for event distribution. Size buffers based on expected throughput (e.g., 100 for event bus). Always include `select` with `default` case or monitor channel capacity to prevent deadlocks.
5. **Metrics Collection**: Use Prometheus histograms with appropriate buckets for timing metrics (e.g., `prometheus.DefBuckets` for seconds). Label metrics with `function_name` and `instance_id` for granular observability.
6. **Graceful Shutdown**: Implement `Shutdown()` methods that first stop accepting new requests, wait for in-flight requests with a timeout, then clean up resources. Use `sync.WaitGroup` to track active goroutines.

Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Request times out immediately	No warm instances and cold start timeout too short	Check warm pool metrics, container startup logs	Increase cold start timeout, configure predictive warming
Instance stuck in PROVISIONING	Container image pull failure or resource constraints	Check container manager logs, docker daemon status	Verify base images exist, increase resource limits
Memory usage grows indefinitely	Instance not being returned to pool after errors	Monitor instance state transitions, check error handling in router	Ensure `ReleaseInstance` is called in all code paths
Queue depth constantly high	Insufficient instances for load	Check scaling metrics, concurrency limits	Adjust scaling thresholds, increase MaxInstances limit
Stale instances in warm pool	TTL/idle timeout not enforced	Check `cleanupExpired` goroutine, instance timestamps	Ensure cleanup goroutine is running, verify time calculations
Response times spike intermittently	Garbage collection in long-running instances	Monitor Go GC pauses, memory allocation patterns	Implement instance rotation, limit instance lifetime

Error Handling and Edge Cases

> **Milestone(s):** Milestone 2 (Execution Environment), Milestone 3 (Cold Start Optimization), Milestone 4 (Request Routing), Milestone 5 (Auto-Scaling)

In a distributed serverless runtime, **errors are not exceptional—they're expected**. Functions execute in ephemeral, isolated environments with strict resource limits, and the control plane must manage thousands of these environments simultaneously. This section catalogs the **failure modes** our system will encounter and the **recovery strategies** that maintain system resilience while meeting service-level objectives. The design follows a **defense-in-depth** approach: each component has local error handling, while the `FlowCoordinator` orchestrates cross-component recovery when local strategies fail.

The Emergency Response Analogy

Think of the runtime as a **city emergency response system**. When a fire (error) breaks out in one building (function instance), the system must:

1. **Detect the emergency quickly** (smoke alarms, 911 calls)
2. **Contain the damage** (firefighters cordon off the building)
3. **Evacuate and reroute** (redirect traffic, provide alternate routes)
4. **Restore normal operations** (repair the building or rebuild it)
5. **Learn and improve** (analyze what caused the fire to prevent recurrence)

Our error handling follows this same pattern: detection → containment → redirection → recovery → learning. Each component has "sensors" (health checks, timeouts, metrics) and "responders" (circuit breakers, retry logic, instance replacement) that work together to maintain system stability.

Failure Modes and Mitigations

The table below categorizes failures by their **failure domain**—whether they occur in the data plane (during function execution) or control plane (during orchestration). Each row specifies the failure, how we detect it, and the automatic recovery action. The **recovery principle** column explains the underlying strategy guiding the specific actions.

Failure Domain	Failure Mode	Description	Detection Mechanism	Recovery Action	Recovery Principle
Function Execution	**Out-of-Memory (OOM) Kill**	Function exceeds its `MemoryMB` limit, triggering Linux OOM killer or container runtime termination	Container exits with SIGKILL (137) or OOM-specific exit code; `ContainerManager.GetContainerStats()` shows abrupt termination; `Container` status changes to `StatusDead`	1. **Immediate**: Record OOM in `Metrics.InvocationErrors` with label `error_type="oom"` 2. **Instance cleanup**: Call `CleanupInstance()` to remove the failed container 3. **Request retry**: For synchronous invocations with `RetryOnColdStart=true`, retry on a fresh instance (max 1 retry for OOM) 4. **Circuit breaker**: Increment failure count in `routedInstance.failures`, possibly tripping circuit breaker	**Fail-fast with controlled retry**: OOMs are likely due to function behavior, so retry cautiously with backpressure
Function Execution	**Timeout Exceeded**	Function execution exceeds configured `Timeout` (wall-clock time)	`Executor.ExecuteFunction()` context deadline exceeded; `InvocationStatus` changes to `InvocationStatusTimeout`; Request `Deadline` passes `TimeRemaining() <= 0`	1. **Terminate execution**: Call `StopContainer()` with `timeout=0` (force kill) 2. **Cleanup**: Remove container via `RemoveContainer()` 3. **No retry by default**: Return timeout error to client (retries would likely timeout again) 4. **Circuit breaker**: Mark instance as unhealthy if multiple timeouts occur	**Graceful degradation**: Timeouts indicate function slowness, not transient failure; avoid retry storms
Function Execution	**Function Crash (Non-zero exit)**	Function process exits with non-zero exit code (application error, panic, unhandled exception)	Container exits with non-zero status; `Container.Status` becomes `StatusExited`; `ExecuteInContainer()` returns error with exit code	1. **Error classification**: Parse exit code to determine if retryable (e.g., temporary file error) or non-retryable (e.g., syntax error) 2. **Retry policy**: For retryable errors, retry on same instance if healthy, otherwise new instance (max 2 retries) 3. **Instance health**: Increment `routedInstance.failures`; if exceeds threshold, mark `healthy=false` 4. **Metrics**: Record in `InvocationErrors` with `error_type="crash"`	**Semantic retry**: Distinguish between transient and permanent failures using exit codes
Function Execution	**Sandbox Escape Attempt**	Function attempts to break out of isolation via privileged syscall, container breakout, or resource exhaustion attack	Seccomp/AppArmor violation (SIGSYS); `ContainerManager` logs security violation; abnormal resource usage patterns in `ContainerStats`	1. **Immediate termination**: Kill container via `StopContainer()` with `timeout=0` 2. **Blacklist**: Optionally blacklist function version if repeated violations 3. **No retry**: Fail request immediately with security violation error 4. **Alert**: Raise security alert to operator	**Zero-tolerance containment**: Security violations are non-retryable and require operator investigation
Function Execution	**Resource Leak (File descriptors, threads)**	Function exhausts per-container resources without triggering OOM	`ContainerStats.PIDs` approaches limit; `GetContainerStats()` shows high FD usage; subsequent invocations fail with "resource unavailable" errors	1. **Proactive recycling**: `WarmPoolManager` calls `CleanupInstance()` between invocations to reset environment 2. **Health checks**: Periodic `ExecuteInContainer()` with	**Preventive maintenance**: Assume functions are buggy; proactively clean and

Failure Domain	Failure Mode	Description	Detection Mechanism	Recovery Action	Recovery Principle
				simple "ping" command to verify instance health before reuse 3. **Instance replacement**: If resource usage exceeds 80% of limit, mark instance for replacement after current invocation	validate between uses
Control Plane	**Cold Start Timeout**	Creating/initializing a new instance exceeds acceptable latency (e.g., >10 seconds)	`RecordColdStart()` measures time from `EventCreateInstance` to `EventContainerStarted`; `Metrics.ColdStartLatency` exceeds threshold; `InstanceState` stuck in `InstanceStateProvisioning`	1. **Abandon creation**: Cancel the pending container creation context 2. **Retry on alternate node**: For multi-node deployments, retry creation on different worker 3. **Fallback to warm**: If available, use existing warm instance even if not ideal (e.g., different version) 4. **Queue timeout**: If no instance ready within timeout, fail request with "cold start timeout"	**Time-bound provisioning**: Don't let slow provisioning block request processing indefinitely
Control Plane	**Container Startup Failure**	Container fails to start due to missing base image, invalid command, or runtime conflict	`ContainerManager.CreateContainer()` or `StartContainer()` returns error; `Container.Status` becomes `StatusDead` without reaching `StatusRunning`	1. **Exponential backoff**: Delay retry with exponential backoff (1s, 2s, 4s...) 2. **Image pre-pull**: Ensure base images are pre-pulled to worker nodes 3. **Fallback image**: Use older compatible runtime image if latest fails 4. **Alert**: Notify operator of persistent image pull failures	**Degraded operation**: Continue serving other functions while diagnosing image issues
Request Routing	**Instance Unhealthy (Zombie)**	Instance appears healthy but doesn't respond to requests (network partition, hung process)	Health check fails (TCP connection refused, HTTP 5xx); `forwardRequest()` timeout; `CircuitBreaker` trips after consecutive failures	1. **Circuit breaker**: Trip circuit for that instance; stop routing requests to it 2. **Replacement**: Schedule instance for termination via `EventForceTerminate` 3. **Drain existing**: If instance has `activeReqs > 0`, wait `DrainTimeout` before force termination 4. **New instance**: Trigger scale-up to replace lost capacity	**Failover with isolation**: Isolate failed instance before removing it to avoid interrupting in-flight requests
Request Routing	**Request Queue Overflow**	Request queue reaches `maxSize`, unable to accept new requests	`RequestQueue.Enqueue()` returns "queue full" error; `gatewayMetrics.queueDepth` at 100% capacity	1. **Load shedding**: Reject new requests with HTTP 503 "Service Unavailable" 2. **Priority-based eviction**: If using priority queue, evict lowest-priority requests first 3. **Emergency scale-up**: Bypass cooldown to scale up immediately 4. **Client backoff**: Include `Retry-After` header in 503 response	**Graceful load shedding**: Better to reject some requests than fail all due to overload
Request Routing	**Gateway Timeout**	Entire request path exceeds `GatewayTimeout` before reaching function	`Gateway` middleware cancels context after timeout; `InvocationStatus` remains `InvocationStatusPending`	1. **Cancel downstream**: Cancel context for all downstream operations (queue, routing, execution) 2. **Cleanup**: If request was dequeued but not yet assigned, return to queue (if idempotent)	**Holistic timeout propagation**: All components respect the

Failure Domain	Failure Mode	Description	Detection Mechanism	Recovery Action	Recovery Principle
				3. **Client response**: Return HTTP 504 "Gateway Timeout" 4. **Avoid retry storm**: Client should use exponential backoff after gateway timeout	same timeout chain
Auto-Scaling	**Scaling Thrashing**	Rapid oscillation between scale-up and scale-down due to metric noise or misconfigured thresholds	`Scaler` metrics show frequent `ScaleDecision` changes; `ScaleUp` immediately followed by `ScaleDown`; high `ContainerOperations` churn rate	1. **Hysteresis**: Use different thresholds for scale-up vs scale-down (e.g., scale-up at 70% concurrency, scale-down at 30%) 2. **Cooldown periods**: Enforce `ScaleUpCooldown` and `ScaleDownCooldown` between scale actions 3. **Smoothing**: Use moving averages instead of instantaneous metrics 4. **Minimum instance lifetime**: Ensure instances live for at least `InstanceTTL` before scale-down	**Stability over optimality**: Prefer slightly suboptimal scaling to constant churn
Auto-Scaling	**Scale-to-Zero Race Condition**	Last instance terminates while new request arrives	`InstanceState` transitions to `InstanceStateTerminated` while request in flight or queued	1. **Termination delay**: When scaling to zero, wait `IdleTimeout + buffer` before terminating last instance 2. **Recreate on demand**: If request arrives after termination, trigger cold start 3. **Request preservation**: Maintain queue across scale-to-zero transitions 4. **Idle instance promotion**: Keep one instance in `InstanceStateWarm` if requests pending	**Lazy termination**: Verify true idle state before terminating last instance
Data Persistence	**Artifact Storage Unavailable**	Backend storage (S3, filesystem) fails during function package upload or retrieval	`ProcessUpload()` returns storage error; `GetArtifact()` fails; `ArtifactStorage` health check fails	1. **Retry with backoff**: Retry storage operations with exponential backoff 2. **Local cache**: Use local filesystem cache for frequently accessed artifacts 3. **Degraded upload**: Accept uploads but mark as "pending storage" 4. **Read-only mode**: If storage completely unavailable, serve existing functions but reject new deployments	**Caching with eventual consistency**: Prefer serving stale artifacts over complete failure
Lifecycle Management	**Drain Stuck**	Instance in `InstanceStateDraining` never completes (hung request, infinite loop)	`Instance` remains in `InstanceStateDraining` beyond `DrainTimeout`; `activeReqs` stays >0	1. **Force termination**: After `DrainTimeout`, send `EventForceTerminate` 2. **Lossy shutdown**: Kill container via `StopContainer()` with `timeout=0` 3. **Client notification**: If possible, notify client of interrupted request 4. **Post-mortem**: Log container state before termination for debugging	**Bounded graceful period**: Graceful shutdown has a time limit

Failure Domain	Failure Mode	Description	Detection Mechanism	Recovery Action	Recovery Principle
Multi-Tenancy	**Noisy Neighbor**	One function monopolizes shared resources (CPU, I/O), affecting others	`ContainerStats` shows sustained high CPU/IO for one container while others starve; elevated `InvocationDuration` for unaffected functions	1. **Resource enforcement**: Strict `CPUQuota` via cgroups CPU shares 2. **I/O throttling**: Use blkio cgroup to limit disk I/O 3. **Workload isolation**: Schedule aggressive functions on separate worker nodes 4. **SLA enforcement**: Preemptively throttle or reject requests from misbehaving functions	**Strong isolation boundaries**: Assume worst-case behavior and enforce hard limits
Network	**Network Partition**	Worker node loses connectivity to control plane or storage	Health checks fail; `ContainerManager.HealthCheck()` returns error; metrics stop flowing	1. **Fencing**: Control plane marks node as unhealthy and stops routing to it 2. **Self-termination**: Worker node self-terminates its containers after grace period 3. **Reconciliation**: Control plane recreates lost instances on healthy nodes 4. **Client redirection**: Update DNS/routing to bypass partitioned node	**Fail-stop then rebuild**: Assume partitioned nodes are lost and rebuild elsewhere

Error Recovery Architecture

The table above shows individual failure modes, but the system's resilience comes from **coordinated recovery across components**. The architecture follows these principles:

1. **Defense in Depth**: Each component handles local errors, with escalation to `FlowCoordinator` for cross-component issues
2. **Graceful Degradation**: When optimal operation isn't possible, provide degraded but functional service
3. **Observability-Driven**: All errors are captured in metrics and logs for analysis and improvement
4. **Idempotent Operations**: Recovery actions can be safely retried without double side effects

The `FlowCoordinator` serves as the **central nervous system** for error recovery. It subscribes to `InstanceEvent` streams from all components and implements the following recovery state machine:

Current State	Trigger Event	Recovery Action	Next State
`InstanceHealthy`	`EventHealthCheckFailed` (1x)	Mark instance `healthy=false` in router	`InstanceUnhealthy`
`InstanceUnhealthy`	`EventHealthCheckFailed` (3x)	Trip circuit breaker; schedule replacement	`InstanceReplacing`
`InstanceReplacing`	`EventCreateInstance` (new)	Route traffic to new instance	`InstanceHealthy`
`InstanceReplacing`	`EventForceTerminate` (old)	Force kill old container	`InstanceTerminated`
`ScaleOscillation`	Consecutive `ScaleUp` / `ScaleDown`	Increase cooldown periods; log alert	`ScaleStabilized`
`StorageDegraded`	Storage errors > threshold	Enable read-only mode; alert operator	`StorageMaintenance`

Common Error Propagation Patterns

When an error occurs, it propagates through the system following these paths:

1. **Execution Error → Router → Client**

```
Function OOM → Executor detects exit code → Router.forwardRequest() returns error →
Router.updateInstanceHealth() marks instance unhealthy → Gateway returns 500 to client
```

2. **Infrastructure Error → FlowCoordinator → Auto-recovery**

```
Container startup failure → WarmPoolManager emits EventContainerStartFailed →
FlowCoordinator.handleEvent() → Scaler.evaluateScaling() triggers scale-up →
New instance created → Router updates routing table
```

3. **Resource Exhaustion → Multiple Components → Load Shedding**

```
Memory pressure → Multiple OOM events → Metrics show high error rate →
Scaler triggers scale-up → RequestQueue fills → Gateway returns 503 →
Client backs off → System stabilizes
```

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
Error Detection	Exit code parsing, timeout contexts	eBPF probes for deep container inspection, distributed tracing
Health Checking	TCP/HTTP health checks	Custom protocol with version/checksum validation
Circuit Breaking	Simple failure count threshold	Adaptive circuit breaking based on latency percentiles
Retry Logic	Fixed-count retry with jitter	Exponential backoff with retry budgets across function boundaries
Error Metrics	Prometheus counters with error type labels	Structured logging with error causality chains

Error Handling Infrastructure

Create a shared error handling package that standardizes error classification and recovery policies:

```go
// internal/errors/errors.go

package errors

import (
    "context"
    "errors"
    "time"
)

// ErrorType categorizes errors for metrics and recovery decisions
type ErrorType string

const (
    ErrorTypeOOM          ErrorType = "oom"
    ErrorTypeTimeout      ErrorType = "timeout"
    ErrorTypeCrash        ErrorType = "crash"
    ErrorTypeSandbox      ErrorType = "sandbox_escape"
    ErrorTypeNetwork      ErrorType = "network"
    ErrorTypeStorage      ErrorType = "storage"
    ErrorTypeStartup      ErrorType = "startup"
    ErrorTypeHealthCheck  ErrorType = "health_check"
)

// ClassifiedError wraps an error with classification metadata
type ClassifiedError struct {
    Err         error
    Type        ErrorType
    Retryable   bool
    ExitCode    int
    InstanceID  string
    Function    string
    Timestamp   time.Time
}

// RecoveryPolicy determines retry behavior based on error type
type RecoveryPolicy struct {
    MaxRetries      int
    BackoffBase     time.Duration
    BackoffMax      time.Duration
    RetryOnSameInstance bool
}

// PolicyForErrorType returns recovery policy for error type
```

```go
func PolicyForErrorType(t ErrorType) RecoveryPolicy {
    switch t {
    case ErrorTypeOOM:
        return RecoveryPolicy{MaxRetries: 1, BackoffBase: 0, RetryOnSameInstance: false}
    case ErrorTypeTimeout:
        return RecoveryPolicy{MaxRetries: 0} // No retry for timeouts
    case ErrorTypeCrash:
        return RecoveryPolicy{MaxRetries: 2, BackoffBase: 100 * time.Millisecond, RetryOnSameInstance: false}
    case ErrorTypeNetwork:
        return RecoveryPolicy{MaxRetries: 3, BackoffBase: 1 * time.Second, RetryOnSameInstance: true}
    default:
        return RecoveryPolicy{MaxRetries: 1, BackoffBase: 0}
    }
}

// ShouldRetry determines if error should be retried based on policy and attempt count
func ShouldRetry(err *ClassifiedError, attempt int) bool {
    policy := PolicyForErrorType(err.Type)
    return err.Retryable && attempt < policy.MaxRetries
}
```

Enhanced Executor with Error Classification

```go
// internal/executor/executor.go (partial)

package executor


import (

    "context"

    "syscall"

    "github.com/ourproject/internal/errors"

)


// executeWithRecovery wraps function execution with error classification and recovery

func (e *Executor) executeWithRecovery(ctx context.Context, def *types.FunctionDefinition, req *types.InvocationRequest)
(*types.InvocationResponse, error) {

    var attempt int

    var lastErr *errors.ClassifiedError



    for attempt = 0; attempt < maxTotalAttempts; attempt++ {

        resp, err := e.executeSingle(ctx, def, req)



        if err == nil {

            return resp, nil

        }



        // Classify the error

        classified := e.classifyExecutionError(err, def, req)

        lastErr = classified



        // Record metrics

        e.metrics.InvocationErrors.WithLabelValues(string(classified.Type), def.Name).Inc()



        // Check if we should retry

        if !errors.ShouldRetry(classified, attempt) {

            break

        }



        // Apply backoff if needed

        policy := errors.PolicyForErrorType(classified.Type)

        if policy.BackoffBase > 0 {

            select {

            case <-ctx.Done():

                return nil, ctx.Err()

            case <-time.After(policy.BackoffBase * (1 << attempt)): // Exponential backoff
```

```
                // Continue to retry
            }
        }

        // If not retrying on same instance, clean up and get new instance
        if !policy.RetryOnSameInstance {
            // TODO: Clean up current container and acquire fresh instance
        }
    }

    // All retries exhausted or non-retryable error
    return nil, &errors.ClassifiedError{
        Err:        fmt.Errorf("execution failed after %d attempts: %v", attempt+1, lastErr.Err),
        Type:       lastErr.Type,
        Retryable:  false,
        InstanceID: lastErr.InstanceID,
        Function:   def.Name,
        Timestamp:  time.Now(),
    }
}

// classifyExecutionError maps raw execution errors to classified types
func (e *Executor) classifyExecutionError(rawErr error, def *types.FunctionDefinition, req *types.InvocationRequest)
*errors.ClassifiedError {
    // TODO 1: Check if error is context deadline exceeded → ErrorTypeTimeout
    // TODO 2: Parse exit code: 137 (SIGKILL) → ErrorTypeOOM
    // TODO 3: Check seccomp violation → ErrorTypeSandbox
    // TODO 4: Check network errors → ErrorTypeNetwork
    // TODO 5: Default to ErrorTypeCrash for other non-zero exits
    // TODO 6: Set Retryable based on error type and exit code
    // TODO 7: Extract instance ID from context or request metadata
    return &errors.ClassifiedError{Type: errors.ErrorTypeCrash, Retryable: false}
}
```

FlowCoordinator Event Handler

```go
// internal/coordinator/flow_coordinator.go (partial)

func (fc *FlowCoordinator) handleEvent(ctx context.Context, event lifecycle.InstanceEvent) error {

    fc.mu.Lock()

    defer fc.mu.Unlock()



    switch event.Type {

    case lifecycle.EventHealthCheckFailed:

        // TODO 1: Get instance from router's routing table

        // TODO 2: Increment failure count for the instance

        // TODO 3: If failure count > threshold, trip circuit breaker

        // TODO 4: Schedule instance replacement via Scaler

        // TODO 5: Emit EventForceTerminate if instance is unrecoverable



    case lifecycle.EventRequestFailed:

        // TODO 1: Parse error type from event.Data

        // TODO 2: Update instance health in router

        // TODO 3: If error indicates resource exhaustion, trigger scale-up

        // TODO 4: Record failure in Metrics.InvocationErrors



    case lifecycle.EventContainerStarted:

        // TODO 1: Update instance state to InstanceStateWarm

        // TODO 2: Register instance with router

        // TODO 3: Update warm pool metrics



    case lifecycle.EventTTLExpired:

        // TODO 1: Check if instance has active requests (activeReqs > 0)

        // TODO 2: If no active requests, terminate immediately

        // TODO 3: If has active requests, transition to InstanceStateDraining

        // TODO 4: Set drain timeout

    }



    return nil

}
```

Debugging Error Scenarios

Symptom	Likely Cause	How to Diagnose	Fix
Functions timeout immediately	Cold start latency exceeds timeout; misconfigured timeout values	Check `Metrics.ColdStartLatency` percentiles; verify function `Timeout` configuration	Increase function timeout; improve cold start optimization; use provisioned concurrency
High OOM error rate	Memory limits too low; function has memory leak	Check `ContainerStats.MemoryUsage` before OOM; compare with `MemoryMB` limit	Increase memory limit; add memory profiling to functions; implement garbage collection between invocations
Request queue constantly full	Insufficient instances; scaling too slow	Check `queueDepth` metrics; examine `Scaler` decisions and cooldown periods	Adjust scaling thresholds; reduce cooldown periods; increase max instances
Instance churn (high create/delete rate)	Scaling thrashing; idle timeout too short	Check `ContainerOperations` counter; examine scale decision log for rapid oscillations	Implement hysteresis; increase cooldown periods; adjust scale thresholds
Functions return "connection refused"	Instance died but still in routing table; network partition	Check instance health status in router; verify container actually running	Implement active health checks; add circuit breakers; reduce health check interval
Cold starts take >10 seconds	Large dependencies; slow storage; image pull latency	Profile cold start phases: image pull, container create, runtime init, handler load	Use pre-warmed base images; implement dependency caching; use snapshot/restore

Testing Error Recovery

Create integration tests that simulate failures and verify recovery:

```bash
# Test OOM recovery
$ curl -X POST http://localhost:8080/invoke/leaky-function \
  -H "Content-Type: application/json" \
  -d '{"action": "allocate_memory", "mb": 1024}'
# Should return 500 on first attempt, then potentially succeed on retry
# Verify metrics show ErrorTypeOOM increment

# Test timeout handling
$ curl -X POST http://localhost:8080/invoke/slow-function \
  -H "Content-Type: application/json" \
  -d '{"sleep_ms": 10000}' \
  --max-time 5
# Should return 504 Gateway Timeout after 5 seconds
# Verify instance is marked unhealthy after multiple timeouts

# Test circuit breaker
$ for i in {1..10}; do
    curl -X POST http://localhost:8080/invoke/unhealthy-function
  done
# First few should return 500, then circuit should trip and return 503
# Verify circuit breaker metrics
```

Testing Strategy

> **Milestone(s):** Milestone 1, Milestone 2, Milestone 3, Milestone 4, Milestone 5

The testing strategy for the serverless function runtime follows a **layered pyramid approach**, emphasizing unit tests at the component level, integration tests for cross-component interactions, and end-to-end system tests for complete workflows. Given the system's distributed nature and reliance on Linux kernel features (cgroups, namespaces), we employ a combination of in-process testing for business logic and container-based testing for isolation features. The testing philosophy prioritizes **deterministic behavior** for core routing and scaling logic while acknowledging **probabilistic outcomes** for race conditions in concurrent scenarios.

Testing Philosophy and Levels

The testing approach is organized into four distinct levels, each serving a specific verification purpose:

1. **Unit Tests** - Test individual components in isolation using mocks and fakes for dependencies
2. **Integration Tests** - Test interactions between 2-3 components with real implementations
3. **System Tests** - Test complete workflows using containerized test environments
4. **Property-Based Tests** - Test invariants and edge cases using generated inputs

The testing pyramid distribution targets 70% unit tests, 20% integration tests, and 10% system tests. Property-based tests supplement each level to uncover edge cases that example-based testing might miss.

Unit Testing Strategy

Unit tests focus on **pure business logic** without external dependencies. Each component's public API is tested against a comprehensive matrix of inputs, including error conditions. The table below outlines the key unit testing patterns:

Component	Testing Focus	Mock Dependencies	Key Assertions
`Router`	Routing logic, load balancing, circuit breaking	`InstanceManager`, network calls	Instance selection correctness, circuit breaker state transitions
`Scaler`	Scaling decisions, metrics aggregation	`MetricsCollector`, `WarmPoolManager`	Scale decision triggers, cooldown enforcement, threshold calculations
`WarmPoolManager`	Pool management, TTL enforcement	`ContainerManager`, time source	Pool size limits, instance reuse, cleanup scheduling
`Coordinator` (packaging)	Dependency resolution, artifact building	Filesystem, network calls	Artifact integrity, dependency conflict handling
`Executor`	Container lifecycle, resource limits	`ContainerManager`, cgroup operations	Resource limit application, timeout enforcement

Each unit test follows the **Arrange-Act-Assert** pattern with clear separation of test setup, execution, and verification. Mock implementations use the Go `interface` types defined in the architecture, allowing dependency injection.

Integration Testing Strategy

Integration tests verify that components work correctly together, particularly focusing on:

- **Lifecycle coordination** between router, warm pool, and scaler
- **Error propagation** across component boundaries
- **Concurrent access patterns** with realistic timing
- **State consistency** during failure recovery

Integration tests use **real implementations** for adjacent components but may mock outbound dependencies like Docker daemon or storage backends. The test environment provides **deterministic time** via a mockable clock interface to eliminate timing flakiness.

System Testing Strategy

System tests execute complete workflows in an environment that approximates production:

- **Container-based test runner** that creates isolated Docker networks
- **Function simulation** using lightweight HTTP servers in containers
- **Load generation** to test scaling behavior under realistic patterns
- **Failure injection** to verify recovery mechanisms

System tests are **resource-intensive** and run in CI/CD pipelines rather than developer workstations. They use the `testcontainers` pattern to spin up necessary dependencies (Docker daemon, mock storage) and clean them up after test completion.

Test Environment Architecture

The test environment is structured to support all testing levels while maintaining reproducibility:

```
test-environment/
├── mocks/                  # Generated mock implementations
├── fixtures/               # Test data and function code samples
├── integration/            # Integration test suites
├── system/                 # System test suites
└── helpers/                # Reusable test utilities
```

Key test utilities include:

Utility	Purpose	Used In
`MockContainerManager`	Simulates container operations without Docker	Unit & integration tests
`FakeClock`	Provides controllable time for timing-sensitive tests	All test levels
`RecordingMetrics`	Captures metric emissions for verification	Unit & integration tests
`TestFunctionServer`	HTTP server simulating function behavior	Integration & system tests
`InMemoryStorage`	Ephemeral storage for artifacts and metadata	Unit & integration tests

Testing Edge Cases and Failure Modes

The testing strategy explicitly addresses the failure modes documented in the "Error Handling and Edge Cases" section. Each failure mode has corresponding test scenarios:

Failure Mode	Test Category	Verification Approach
Container OOM kill	System test	Configure memory limit below function requirement, verify error classification
Cold start timeout	Integration test	Mock slow container startup, verify retry logic
Scaling thrashing	Property-based test	Generate oscillating load patterns, verify cooldown prevents thrashing
Queue overflow	Integration test	Send requests exceeding queue capacity, verify load shedding
Network partition	System test	Simulate network failure between components, verify circuit breaking
Resource exhaustion	Integration test	Exhaust file descriptors, CPU, memory; verify graceful degradation

Property-based tests using the `quick` or `gopter` libraries generate random inputs to discover edge cases in state transitions, especially in the `StateMachine` and `Router` components.

Performance and Benchmark Testing

Beyond functional correctness, the system includes performance benchmarks to track:

- **Cold start latency** under various configurations
- **Memory footprint** of warm pool instances
- **Routing overhead** at different request rates
- **Scaling decision latency** during load spikes

Benchmarks use the Go `testing.B` framework and produce comparative reports. Performance regressions trigger alerts in the CI pipeline.

Test Data Management

Test data follows these principles:

- **Minimal representative samples** for function code (hello-world handlers)
- **Generated test cases** for dependency resolution scenarios
- **Golden files** for expected artifact structures
- **Seeded randomness** for reproducible property tests

Function test fixtures include examples for each supported runtime (Go, Python, Node.js) with varying complexity levels.

Continuous Integration Pipeline

The CI pipeline executes tests in this sequence:

1. **Static analysis** (go vet, staticcheck, security scanning)
2. **Unit tests** (fast, mandatory for all PRs)
3. **Integration tests** (medium duration, required for main branch)

4. **System tests** (long duration, nightly runs)
5. **Benchmarks** (performance tracking, weekly runs)

Test results include coverage reports, with a minimum threshold of 80% line coverage for the core components (`Router` , `Scaler` , `WarmPoolManager` , `Executor`).

Milestone Implementation Checkpoints

Each milestone has specific verification checkpoints to ensure incremental progress. The checkpoints progress from isolated component behavior to integrated system behavior.

Milestone 1: Function Packaging Checkpoints

> **Objective:** Verify that function code can be uploaded, packaged with dependencies, stored, and retrieved.

Checkpoint	Expected Behavior	Verification Commands	Success Criteria
1.1 - Upload API	HTTP endpoint accepts multipart file upload and returns function ID	`curl -X POST -F "file=@hello.go" http://localhost:8080/upload`	Returns JSON with `{"function_name": "hello", "version": "v1", "artifact_hash": "sha256:..."}`
1.2 - Dependency Resolution	Go dependencies resolved from `go.mod` and bundled	Upload function with imports; inspect generated artifact	Artifact contains all transitive dependencies in `vendor/` directory
1.3 - Artifact Storage	Package stored durably and retrievable by hash	Upload function, then `curl http://localhost:8080/artifacts/{hash}`	Returns exact same bytes as uploaded (verified by hash comparison)
1.4 - Multi-Runtime Support	Python and Node.js functions packaged correctly	Upload Python function with `requirements.txt` , Node.js with `package.json`	Python artifact includes virtualenv, Node.js includes `node_modules`
1.5 - Version Immutability	Same code produces same hash, preventing modification	Upload identical function twice	Same artifact hash returned; storage reports duplicate
1.6 - Validation	Invalid function definitions rejected	Upload function without handler, with unsupported runtime	Returns 400 Bad Request with specific error message

Verification workflow:

```bash
# Run unit tests for packaging components
go test ./internal/packaging/... -v -count=1

# Start test server with in-memory storage
go run cmd/test-server/main.go --storage=memory

# Upload test function (from testdata directory)
curl -X POST -F "file=@testdata/hello-go/hello.go" \
  -F "runtime=go" -F "handler=Handler" \
  http://localhost:8080/upload

# Verify artifact exists
curl http://localhost:8080/artifacts/{hash-from-response}
```

Expected observations:

- Upload endpoint responds within 500ms for simple functions
- Artifact size correlates with dependency count (e.g., empty function ~10KB, with dependencies ~1-10MB)
- Hash changes when any file content changes (code or dependencies)
- Storage backend reports deduplication for identical uploads

Milestone 2: Execution Environment Checkpoints

> **Objective:** Verify that functions execute in isolated containers with enforced resource limits.

Checkpoint	Expected Behavior	Verification Commands	Success Criteria
2.1 - Container Creation	Isolated environment created from base image	`executor.ExecuteFunction` with simple function	Container created with unique ID, runtime-specific base image
2.2 - Resource Limits	Memory and CPU limits enforced	Function that allocates memory beyond limit	Container terminated with OOM kill, `ErrorTypeOOM` classification
2.3 - Timeout Enforcement	Execution terminated after timeout	Function with `time.Sleep` exceeding timeout	Returns `ErrorTypeTimeout` after configured duration
2.4 - Filesystem Isolation	Clean `/tmp` per invocation	Function writes to `/tmp/test.txt`	File exists during execution, not visible to other invocations
2.5 - Network Isolation	Outbound connectivity controlled	Function attempts to connect to external service	Connection succeeds/fails per network policy configuration
2.6 - Cleanup	Resources released after execution	Monitor Docker containers before/after execution	No lingering containers from completed executions

Verification workflow:

```bash
# Test resource limit enforcement
go test ./internal/executor/... -run TestMemoryLimit -v

# Run integration test with actual Docker
DOCKER_HOST=unix:///var/run/docker.sock \
  go test ./integration/executor/... -v

# Manual test: run memory-hungry function
cat > memory_test.go << 'EOF'
package main
func Handler() string {
  // Allocate 200MB when limit is 128MB
  data := make([]byte, 200*1024*1024)
  return "ok"
}
EOF

# Upload and invoke with 128MB memory limit
# Expect OOM error in response
```

Expected observations:

- Container creation takes 500ms-2s depending on base image size
- Memory limit violations detected within 100ms of allocation
- Timeout precision within ±50ms of configured value
- Post-execution Docker `ps` shows no containers with project labels
- CPU throttling observable via `docker stats` during CPU-intensive functions

Milestone 3: Cold Start Optimization Checkpoints

> **Objective:** Verify that warm instances reduce startup latency and snapshot/restore works correctly.

Checkpoint	Expected Behavior	Verification Commands	Success Criteria
3.1 - Warm Pool Population	Instances created and added to pool	`PreWarm` called for function	Specified number of containers in `StatusRunning` but idle
3.2 - Warm Start Latency	Request to warm instance faster than cold	Measure `AcquireWarmInstance` + execution vs cold path	Warm start < 100ms, cold start > 500ms (language-dependent)
3.3 - Instance Reuse	Instance returns to pool after execution	Invoke function twice with `ReleaseInstance`	Same container ID used for both invocations
3.4 - Pool Size Limits	Pool respects max/min instance counts	Call `PreWarm` beyond max, monitor pool	Extra instances not created or immediately cleaned up
3.5 - TTL Expiration	Old instances removed from pool	Create instance, wait beyond TTL, check cleanup	Instance terminated, removed from pool store
3.6 - Snapshot/Restore	Checkpointed container restores quickly	Use CRIU to checkpoint, measure restore time	Restore < 50ms from snapshot file

Verification workflow:

```bash
# Benchmark cold vs warm starts
go test ./internal/pool/... -bench=".*Start.*" -benchtime=10s

# Test pool lifecycle
go test ./internal/pool/... -run TestPoolLifecycle -v

# Manual test: observe container reuse
# Terminal 1: Start gateway with verbose logging
LOG_LEVEL=debug go run cmd/gateway/main.go

# Terminal 2: Invoke function twice with 1s gap
curl http://localhost:8080/invoke/hello
sleep 1
curl http://localhost:8080/invoke/hello

# Check logs: second request should show "using warm instance"
```

Expected observations:

- First invocation shows "cold start" in logs with 500ms+ duration
- Second invocation shows "warm start" with <100ms duration
- `docker ps` shows container persists between invocations
- Pool metrics show instance count stabilizing at configured max/min
- Memory usage increases with pool size but stabilizes
- Old instances cleaned up exactly at TTL expiration

Milestone 4: Request Routing Checkpoints

> **Objective:** Verify that requests are correctly routed, queued, load balanced, and timed out.

Checkpoint	Expected Behavior	Verification Commands	Success Criteria
4.1 - HTTP Gateway	Endpoint accepts requests, routes to function	`curl http://localhost:8080/invoke/hello`	Request forwarded to function instance, response returned
4.2 - Load Balancing	Requests distributed across instances	Launch 3 instances, send 10 requests	Requests spread across instances (round-robin or least-connections)
4.3 - Request Queuing	Queue buffers requests when busy	All instances busy, send additional request	Request queued, executes when instance available
4.4 - Concurrency Limits	Per-function concurrency enforced	Send parallel requests exceeding limit	Excess requests queued or rejected (per configuration)
4.5 - Circuit Breaking	Unhealthy instances avoided	Kill container, send requests	Failures counted, instance marked unhealthy after threshold
4.6 - Timeout Propagation	Deadline passed through layers	Function sleeps beyond timeout	Request cancelled at all levels with `InvocationStatusTimeout`

Verification workflow:

```bash
# Test routing logic
go test ./internal/gateway/... -run TestRouter -v

# Start gateway with test configuration
go run cmd/gateway/main.go --config testdata/gateway-config.yaml

# Load test with multiple concurrent requests
# Using hey or vegeta
echo '{"message": "test"}' | vegeta attack \
  -duration=10s -rate=10 -targets=targets.txt \
  | vegeta report

# Test queue behavior by saturating instances
# Terminal 1: Slow function
cat > slow.go << 'EOF'
package main
import "time"
func Handler() string {
  time.Sleep(5 * time.Second)
  return "done"
}
EOF

# Terminal 2: Send burst of requests
for i in {1..10}; do
  curl http://localhost:8080/invoke/slow &
done

# Observe: first N execute immediately (N = concurrency limit),
# rest queue, execute as instances free up
```

Expected observations:

- Gateway responds to health check at `/health`
- Round-robin distribution visible in instance assignment logs
- Queue depth metric increases when all instances busy
- Circuit breaker trips after 5 consecutive failures (default)
- Request cancellation propagates within 100ms of timeout
- Sticky routing (if enabled) sends same client to same instance

Milestone 5: Auto-Scaling Checkpoints

> **Objective:** Verify that instances scale up/down based on load and scale to zero after inactivity.

Checkpoint	Expected Behavior	Verification Commands	Success Criteria
5.1 - Metrics Collection	Invocation rate and concurrency tracked	Send requests at varying rates	`GetRequestRate()` returns accurate RPS over sliding window
5.2 - Scale-Up Trigger	New instances created when threshold exceeded	Send requests above target concurrency	New containers created, added to pool/routing table
5.3 - Scale-Down Cooldown	Instances not removed too quickly	Spike traffic, then drop to zero, monitor	Scale-down waits for cooldown period before removing instances
5.4 - Scale-to-Zero	All instances removed after idle timeout	No requests for > idle timeout	Container count drops to zero, warm pool empty
5.5 - Provisioned Concurrency	Minimum instances kept warm	Configure min=2, monitor during idle	2 instances remain in warm pool regardless of traffic
5.6 - Scaling Metrics	Scale events recorded in metrics	Scale up/down several times	`scaling_operations_total` counter increments with direction labels

Verification workflow:

```bash
# Test scaling logic
go test ./internal/scaler/... -run TestScaleDecision -v

# Integration test with simulated traffic
go test ./integration/scaler/... -v -timeout=2m

# Manual test: observe scaling behavior
# Start system with auto-scaling enabled
go run cmd/coordinator/main.go --enable-scaling

# Generate load pattern
# Phase 1: Low traffic (should scale to min or zero)
sleep 30

# Phase 2: Burst traffic (should scale up)
hey -z 30s -q 10 http://localhost:8080/invoke/hello

# Phase 3: Steady moderate traffic (should maintain)
hey -z 60s -q 5 http://localhost:8080/invoke/hello

# Phase 4: No traffic (should scale down after cooldown)
sleep 90

# Monitor scaling events in logs and metrics endpoint
curl http://localhost:8080/metrics | grep scaling_
```

Expected observations:

- Scale-up occurs within 10-30 seconds of sustained load (configurable)
- Scale-down respects cooldown period (default 300s)
- Scale-to-zero removes last instance after idle timeout (default 600s)
- Provisioned concurrency keeps instances warm even with no traffic
- Scaling decisions logged with rationale (e.g., "scale up: concurrency 12 > threshold 10")
- No thrashing: stable traffic produces stable instance count

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Unit Testing	Standard `testing` package + `testify/assert`	`ginkgo` / `gomega` BDD framework
Mock Generation	Manual interface implementations	`mockery` or `moq` code generation
Integration Testing	Docker SDK + testcontainers	Kubernetes in Docker (kind) cluster
Property Testing	`testing/quick`	`gopter` with custom generators
Benchmarking	`testing.B` with `-bench`	Custom benchmark harness with percentile metrics
Coverage Analysis	`go test -cover`	`goverage` or `codecov` integration
Load Testing	`vegeta` command line	Custom load generator with scenario modeling

B. Recommended File/Module Structure for Testing

```
serverless-runtime/
├── cmd/
│   ├── test-server/           # Dedicated test server
│   │   └── main.go
│   └── load-generator/        # Load testing tool
│       └── main.go
├── internal/
│   ├── gateway/
│   │   ├── gateway.go
│   │   ├── gateway_test.go    # Unit tests
│   │   └── integration_test.go # Integration tests
│   ├── scaler/
│   │   ├── scaler.go
│   │   ├── scaler_test.go
│   │   └── benchmark_test.go  # Benchmarks
│   └── ... (other components)
├── test/
│   ├── mocks/                 # Generated mocks
│   │   ├── container_manager_mock.go
│   │   └── metrics_mock.go
│   ├── fixtures/              # Test data
│   │   ├── functions/
│   │   │   ├── hello-go/
│   │   │   ├── hello-python/
│   │   │   └── hello-node/
│   │   └── configs/
│   ├── integration/           # Integration test suites
│   │   ├── setup.go
│   │   ├── test_router_scaler.go
│   │   └── test_pool_lifecycle.go
│   ├── system/               # System tests
│   │   ├── docker_test.go
│   │   └── scaling_test.go
│   └── helpers/              # Test utilities
│       ├── clock.go
│       ├── container_helper.go
│       └── metrics_helper.go
├── go.mod
└── go.sum
```

C. Infrastructure Starter Code for Test Utilities

`test/helpers/clock.go` - Deterministic time for tests:

```go
package helpers

import (
    "sync"
    "time"
)

// FakeClock provides deterministic time control for tests
type FakeClock struct {
    mu      sync.RWMutex
    current time.Time
    timers  []*fakeTimer
}

// NewFakeClock creates a FakeClock at the given time
func NewFakeClock(initial time.Time) *FakeClock {
    return &FakeClock{current: initial}
}

// Now returns the current fake time
func (c *FakeClock) Now() time.Time {
    c.mu.RLock()
    defer c.mu.RUnlock()
    return c.current
}

// Advance moves time forward by duration
func (c *FakeClock) Advance(d time.Duration) {
    c.mu.Lock()
    defer c.mu.Unlock()
    c.current = c.current.Add(d)

    // Trigger any timers that have expired
    for _, t := range c.timers {
        t.check(c.current)
    }
}

// AfterFunc mimics time.AfterFunc
func (c *FakeClock) AfterFunc(d time.Duration, f func()) *fakeTimer {
    c.mu.Lock()
    defer c.mu.Unlock()
```

```
    t := &fakeTimer{
        expiry: c.current.Add(d),
        f:      f,
        active: true,
    }
    c.timers = append(c.timers, t)
    return t
}

type fakeTimer struct {
    expiry time.Time
    f      func()
    active bool
}

func (t *fakeTimer) check(now time.Time) {
    if t.active && !now.Before(t.expiry) {
        t.active = false
        go t.f()
    }
}
```

`test/helpers/metrics_helper.go` - **Metrics recording for verification:**

```go
package helpers

import (
    "sync"
    "github.com/prometheus/client_golang/prometheus"
)

// RecordingMetrics captures metric emissions for test verification
type RecordingMetrics struct {
    mu sync.RWMutex

    // Capture cold start latencies
    ColdStartLatencies []float64

    // Capture invocation counts by function
    InvocationCounts map[string]int

    // Capture error counts by type
    ErrorCounts map[string]map[string]int
}

// NewRecordingMetrics creates a new RecordingMetrics
func NewRecordingMetrics() *RecordingMetrics {
    return &RecordingMetrics{
        InvocationCounts: make(map[string]int),
        ErrorCounts:      make(map[string]map[string]int),
    }
}

// RecordColdStart captures a cold start duration
func (rm *RecordingMetrics) RecordColdStart(duration time.Duration) {
    rm.mu.Lock()
    defer rm.mu.Unlock()
    rm.ColdStartLatencies = append(rm.ColdStartLatencies, duration.Seconds()*1000) // Convert to ms
}

// RecordInvocation captures an invocation completion
func (rm *RecordingMetrics) RecordInvocation(functionName string, duration time.Duration, err error) {
    rm.mu.Lock()
    defer rm.mu.Unlock()

    rm.InvocationCounts[functionName]++
```

```
    if err != nil {
        errType := classifyTestError(err)
        if rm.ErrorCounts[functionName] == nil {
            rm.ErrorCounts[functionName] = make(map[string]int)
        }
        rm.ErrorCounts[functionName][errType]++
    }
}

// GetColdStartP95 returns the 95th percentile cold start latency
func (rm *RecordingMetrics) GetColdStartP95() float64 {
    rm.mu.RLock()
    defer rm.mu.RUnlock()

    if len(rm.ColdStartLatencies) == 0 {
        return 0
    }

    // Sort and calculate percentile
    latencies := make([]float64, len(rm.ColdStartLatencies))
    copy(latencies, rm.ColdStartLatencies)
    sort.Float64s(latencies)

    index := int(float64(len(latencies)) * 0.95)
    if index >= len(latencies) {
        index = len(latencies) - 1
    }
    return latencies[index]
}
```

D. Core Logic Skeleton Code for Test Verification

`test/integration/scaling_test.go` **- Integration test for auto-scaling:**

```go
package integration

import (
    "context"
    "testing"
    "time"
    "github.com/stretchr/testify/assert"
    "github.com/stretchr/testify/require"

    "serverless-runtime/internal/scaler"
    "serverless-runtime/internal/pool"
    "serverless-runtime/test/helpers"
)

func TestScaleUpOnHighConcurrency(t *testing.T) {
    // TODO 1: Create mock container manager and metrics collector
    mockCM := helpers.NewMockContainerManager()
    metrics := helpers.NewRecordingMetrics()

    // TODO 2: Initialize warm pool with zero instances
    poolConfig := pool.PoolConfig{
        MaxWarmInstances: 10,
        MinWarmInstances: 0,
        InstanceTTL:      5 * time.Minute,
        IdleTimeout:      10 * time.Minute,
    }
    warmPool, err := pool.NewWarmPoolManager(mockCM, poolConfig, metrics)
    require.NoError(t, err)

    // TODO 3: Initialize scaler with target concurrency of 2
    scalingConfig := scaler.ScalingConfig{
        FunctionName:        "test-function",
        MinInstances:        0,
        MaxInstances:        5,
        TargetConcurrency:   2,
        ScaleUpThreshold:    1.5,  // 150% of target
        ScaleDownThreshold:  0.5,  // 50% of target
        ScaleUpCooldown:     30 * time.Second,
        ScaleDownCooldown:   5 * time.Minute,
    }
```

```
    scaler, err := scaler.NewScaler(metrics, warmPool, scalingConfig)
    require.NoError(t, err)

    ctx := context.Background()

    // TODO 4: Simulate 5 concurrent executions (exceeding threshold)
    for i := 0; i < 5; i++ {
        metrics.RecordInvocation("test-function", 100*time.Millisecond, nil)
        metrics.IncrementConcurrency("test-function")
    }

    // TODO 5: Trigger scaling evaluation
    decision := scaler.EvaluateScaling(ctx, "test-function")

    // TODO 6: Verify scale-up decision
    assert.Equal(t, scaler.ScaleUp, decision.Direction)
    assert.Equal(t, 3, decision.Count) // Expected: ceil(5 current / 2 target) = 3 instances
    assert.Contains(t, decision.Reason, "concurrency")

    // TODO 7: Execute scaling decision
    err = scaler.executeScaling(ctx, decision)
    require.NoError(t, err)

    // TODO 8: Verify new instances created in warm pool
    instances, err := warmPool.ListInstances(ctx, "test-function")
    require.NoError(t, err)
    assert.Len(t, instances, 3)

    // TODO 9: Simulate executions completing
    for i := 0; i < 5; i++ {
        metrics.DecrementConcurrency("test-function")
    }

    // TODO 10: Verify scale-down doesn't happen immediately (cooldown)
    decision2 := scaler.EvaluateScaling(ctx, "test-function")
    assert.Equal(t, scaler.ScaleNone, decision2.Direction)
    assert.Contains(t, decision2.Reason, "cooldown")
}
```

`test/helpers/mock_container_manager.go` **- Mock for container operations:**

```go
package helpers

import (
    "context"
    "sync"
    "time"
    "serverless-runtime/internal/container"
    "serverless-runtime/internal/types"
)

// MockContainerManager simulates container operations for tests
type MockContainerManager struct {
    mu          sync.RWMutex
    containers  map[string]*types.Container
    createDelay time.Duration
    startDelay  time.Duration
    failCreate  bool
    failStart   bool
    opsRecorded []string
}

// CreateContainer simulates container creation
func (m *MockContainerManager) CreateContainer(
    ctx context.Context,
    req types.CreateContainerRequest,
) (*types.Container, error) {
    m.mu.Lock()
    defer m.mu.Unlock()

    m.opsRecorded = append(m.opsRecorded, "CreateContainer")

    if m.failCreate {
        return nil, fmt.Errorf("mock container creation failed")
    }

    // Simulate delay
    if m.createDelay > 0 {
        select {
        case <-time.After(m.createDelay):
        case <-ctx.Done():
            return nil, ctx.Err()
        }
```

```
    }

    containerID := fmt.Sprintf("mock-container-%d", len(m.containers))
    container := &types.Container{
        ID:        containerID,
        Image:     req.Image,
        Status:    types.StatusCreated,
        CreatedAt: time.Now(),
        Labels:    req.Labels,
    }

    m.containers[containerID] = container
    return container, nil
}

// StartContainer simulates container startup
func (m *MockContainerManager) StartContainer(
    ctx context.Context,
    containerID string,
) error {
    // TODO 1: Check if container exists
    // TODO 2: Simulate startup delay if configured
    // TODO 3: Update container status to StatusRunning
    // TODO 4: Record operation in opsRecorded
    // TODO 5: Return mock error if failStart is true
}

// ListContainers returns mock containers
func (m *MockContainerManager) ListContainers(
    ctx context.Context,
    filter types.ListFilter,
) ([]types.Container, error) {
    // TODO 1: Filter containers by labels and status
    // TODO 2: Return matching containers
    // TODO 3: Record operation in opsRecorded
}
```

E. Language-Specific Hints for Go Testing

- **Table-Driven Tests:** Use the pattern with `tt := range tests` for comprehensive input coverage
- **Parallel Execution:** Use `t.Parallel()` judiciously for unit tests, but avoid for integration tests with shared resources
- **Test Helpers:** Create `testHelper` functions that accept `testing.TB` (interface for `testing.T` and `testing.B`)
- **Golden Files:** Use `go:embed` to embed test fixtures directly in test files
- **Benchmark Memory:** Use `b.ReportAllocs()` to track memory allocations in benchmarks

- **Cleanup:** Use `t.Cleanup()` for resource cleanup instead of defer in subtests
- **Error Testing:** Use `errors.Is()` and `errors.As()` for error type assertions
- **Context Testing:** Use `context.WithTimeout` in tests to prevent hangs, but mock time for deterministic behavior

F. Debugging Tips for Test Failures

Symptom	Likely Cause	How to Diagnose	Fix
Test passes locally but fails in CI	Timing differences, race conditions	Run with `-race` flag, add logging with timestamps	Use `FakeClock`, increase timeouts, fix data races
Container cleanup failing	Docker API rate limiting, zombie containers	Check Docker logs, monitor container count	Implement retry with backoff, force remove in cleanup
Metrics showing incorrect values	Race condition in metric updates	Use `-race` detector, add metric value snapshots	Protect metric updates with mutex, use atomic operations
Scaling thrashing in tests	Too sensitive thresholds, no hysteresis	Log scaling decisions with metrics values	Increase cooldown periods, add hysteresis (different up/down thresholds)
Cold start timeout flakiness	Variable container startup time	Measure actual startup times, log each phase	Increase timeout buffer, use faster base images
Queue test deadlock	Unbounded queue growth blocking producers	Monitor queue depth metric, add timeout to enqueue	Implement queue size limits, rejection policy for full queues
Race detector reports data race	Concurrent map access, unprotected field writes	Identify the exact line from race report	Use `sync.Map` or `sync.RWMutex`, copy data before goroutine

G. Milestone Verification Commands Summary

Complete test suite execution:

```bash
# Run all unit tests with race detection
go test ./internal/... -race -count=1 -timeout=5m

# Run integration tests (requires Docker)
DOCKER_HOST=unix:///var/run/docker.sock \
  go test ./test/integration/... -v -timeout=10m

# Run benchmarks
go test ./internal/scaler/... -bench=. -benchtime=5s

# Check test coverage
go test ./internal/... -coverprofile=coverage.out
go tool cover -html=coverage.out -o coverage.html

# Run property-based tests
go test ./internal/router/... -run TestPropertyBased -v
```

Post-implementation verification checklist:

- [ ] All unit tests pass with `-race` detector enabled
- [ ] Integration tests pass with real Docker daemon
- [ ] Benchmarks show no significant performance regressions
- [ ] Test coverage > 80% for core components
- [ ] No flaky tests (run test suite 10 times consecutively)
- [ ] System tests pass in CI environment
- [ ] Load tests demonstrate scaling behavior matches specifications
- [ ] Error injection tests verify recovery mechanisms work

Debugging Guide

> **Milestone(s):** Milestone 1, Milestone 2, Milestone 3, Milestone 4, Milestone 5

Debugging a serverless function runtime is challenging because it's a **distributed system with multiple interacting components**, each with their own failure modes. The ephemeral nature of function instances, the complexity of container isolation, and the dynamic scaling behavior create unique debugging scenarios. This guide provides a systematic approach to diagnosing and fixing common issues during development, organized by observable symptoms. Think of debugging this system as **being a detective at a crime scene**—you have multiple witnesses (logs, metrics, traces), circumstantial evidence (system state), and a crime scene that disappears after the crime (ephemeral containers). Your job is to reconstruct what happened from the available evidence.

Common Bug Symptoms and Fixes

The following tables organize debugging by observable symptom, grouping related symptoms from different components. Each symptom includes the **observable behavior** (what you see), the **likely root cause** (what's probably wrong), and **step-by-step investigation and fix**. Use this as a starting point when encountering issues during development.

Function Packaging and Deployment Issues

These symptoms occur during function upload, packaging, or when functions fail to initialize properly.

Symptom	Likely Cause	Steps to Fix
Function upload fails with "invalid handler" error	The handler path specified in the `FunctionDefinition` doesn't match an actual function in the uploaded code. This is often caused by incorrect handler syntax (e.g., `handler.main` for Go when the function is named `HandleRequest`), or the handler file being missing from the uploaded archive.	1. **Check the handler format**: For Go, the handler should be the package-qualified function name (e.g., `main.Handler`). For Python, it should be `module.function_name`. 2. **Verify the uploaded code**: Use the `ValidateHandler` method in the bundler to confirm the handler exists in the workspace. 3. **Inspect the artifact**: Download the stored artifact and examine its structure—ensure the main file with the handler is present.
Function deployment succeeds but instances fail to start with "dependency resolution failed"	The dependency resolution step during packaging didn't capture all required libraries, or there are version conflicts between transitive dependencies. This is particularly common in Python with `requirements.txt` or Node.js with complex dependency trees.	1. **Examine the bundler logs**: The `ResolveDependencies` method should log the resolved dependency graph. Check for warnings about missing or conflicting versions. 2. **Test the build artifact locally**: Extract the built artifact and run it in a minimal environment to see if dependencies are missing. 3. **Add explicit version pins**: In the function's dependency manifest, add explicit versions for transitive dependencies to avoid conflicts. 4. **Check for platform-specific dependencies**: Ensure dependencies are compatible with the runtime's base image (e.g., Linux wheels for Python).
Function package size is unexpectedly large (>100MB)	The packaging process included unnecessary files (development dependencies, test directories, virtual environments, or large binary files) that bloat the artifact. This increases cold start times and storage costs.	1. **Implement `.faasignore`**: Create an ignore file similar to `.dockerignore` that excludes common development artifacts like `node_modules`, `.git`, `__pycache__`, and test directories. 2. **Use dependency pruning**: For interpreted languages, analyze the actual imports and strip unused dependencies. 3. **Enable artifact compression**: Compress the artifact using gzip or zstd before storage. 4. **Check for duplicated layers**: If using container images, ensure common runtime layers are shared across functions.
Function versioning shows incorrect or duplicate versions	The version generation logic (likely based on content hash) is collision-prone or the storage backend isn't properly isolating versions. This can happen when two different functions produce the same hash (rare) or when the storage layer overwrites existing artifacts.	1. **Verify the hash algorithm**: Use a secure hash (SHA-256) that includes the function code, dependencies, and runtime configuration. 2. **Check for non-deterministic builds**: Ensure builds are reproducible—Go builds should use module mode with fixed versions, Python should use locked dependency files. 3. **Inspect the storage backend**: Ensure the artifact storage uses the full hash as the key and doesn't allow overwrites. 4. **Add a version manifest**: Store a separate manifest mapping human-readable versions (e.g., "v1.2") to content hashes.
Upload API times out for large function packages	The `UploadHandler` is blocking on processing the entire upload before responding, or the dependency resolution/building step takes too long without streaming progress. This makes large functions difficult to deploy.	1. **Implement asynchronous upload processing**: Accept the upload, return an immediate response with a job ID, and process the packaging in the background. 2. **Add progress reporting**: For synchronous uploads, implement chunked uploads and periodic status updates. 3. **Set reasonable timeouts**: Configure the gateway timeout (`GatewayConfig.DefaultTimeout`) to allow for large builds but fail fast on truly stuck uploads. 4. **Optimize dependency resolution**: Cache resolved dependencies across functions to speed up subsequent uploads.

Execution Environment and Isolation Issues

These symptoms relate to container lifecycle, resource limits, and security isolation.

Symptom	Likely Cause	Steps to Fix
Function times out immediately on every invocation	The container is failing to start or the entrypoint command is incorrect, causing the function to exit immediately. The `Executor` then times out waiting for a response. This often happens when the base image doesn't match the runtime or the handler binary is missing.	1. **Check container logs**: The `ContainerManager` should capture container stdout/stderr. Look for "executable not found" or "no such file" errors. 2. **Verify the base image**: Ensure `GetBaseImage` returns the correct image for the runtime (e.g., `golang:alpine` for Go, `python:slim` for Python). 3. **Test the container manually**: Use `docker run` with the same image and command to see if it starts. 4. **Inspect the `CreateContainerRequest`**: Ensure `Cmd` and `WorkingDir` are correctly set for the runtime.
Function exceeds memory limit but isn't terminated	The cgroup memory limit isn't being enforced properly, or the OOM killer is configured incorrectly. This can happen if the `CGroupHelper.SetMemoryLimit` isn't setting both `memory.limit_in_bytes` and `memory.memsw.limit_in_bytes` (for swap), or if the container isn't being added to the correct cgroup.	1. **Verify cgroup configuration**: Check `/sys/fs/cgroup/memory/` (or `/sys/fs/cgroup/system.slice/` for systemd) for the container's cgroup and confirm limits are set. 2. **Test with a memory-hungry function**: Write a test function that allocates memory aggressively and verify it gets OOM-killed. 3. **Check for swap**: If swap is enabled, ensure both memory and swap limits are set. 4. **Update the seccomp profile**: Some older Docker versions require specific seccomp rules for cgroup enforcement.
Function instances accumulate and aren't cleaned up ("zombie containers")	The cleanup logic in `CleanupIdleContainers` isn't running or isn't finding the containers, or the `RemoveContainer` method is failing silently. This leads to resource exhaustion over time.	1. **Check the cleanup schedule**: Ensure the `WarmPoolManager.cleanupTicker` is running and `cleanupExpired` is being called. 2. **Verify container listing**: Use `ListContainers` with appropriate filters to see if the manager can find the containers. 3. **Inspect removal errors**: Add logging to `RemoveContainer` and check for "container not found" or permission errors. 4. **Add a garbage collector**: Implement a periodic sweep that removes any container not tracked in the `WarmPoolManager.store`.
Function can access host resources or other function's files	Insufficient namespace isolation—the container might be running with host network, PID, or filesystem namespaces. This is a critical security issue.	1. **Review `CreateContainerRequest` security options**: Ensure `NetworkMode` is set to `bridge` (not `host`), and `SecurityOpts` include `no-new-privileges:true`. 2. **Verify namespace flags**: Check that the container manager is creating containers with all namespaces (`pid`, `net`, `ipc`, `mnt`, `uts`). 3. **Test isolation**: Write a function that tries to list processes (`ps aux`) or read `/etc/hostname`—it should only see its own isolated environment. 4. **Use a security scanner**: Run tools like `docker-bench-security` to check for misconfigurations.
Function execution is much slower than expected	CPU throttling from cgroups is too aggressive, or the container lacks sufficient CPU shares. The `CPUQuota` might be set too low (e.g., 50,000 for 50% of a CPU core).	1. **Check CPU quota settings**: Verify `CreateContainerRequest.CPUQuota` is reasonable (100,000 = 1 CPU core). Consider increasing for CPU-intensive functions. 2. **Monitor CPU throttling**: Use `GetContainerStats` to see `CPUPercent` and check `/sys/fs/cgroup/cpu/` for `nr_throttled` events. 3. **Adjust CPU period**: The default period is 100ms; decreasing it can provide more fine-grained scheduling but may increase overhead. 4. **Consider CPU pinning**: For performance-critical functions, use CPU sets to dedicate cores.

Cold Start and Warm Pool Issues

These symptoms manifest as high latency for first requests or inefficient resource usage.

Symptom	Likely Cause	Steps to Fix
Cold start latency is consistently >1s despite warm pool	The warm pool is empty or instances are expired before use. This could be due to `PoolConfig.MinWarmInstances` being set to 0, `InstanceTTL` being too short, or the pre-warming logic not triggering.	1. **Check warm pool metrics**: Use `Metrics.ContainerCount` to see how many warm instances exist per function. 2. **Verify TTL configuration**: Ensure `InstanceTTL` is longer than the typical idle period between invocations. 3. **Test pre-warming**: Invoke the function once, then check if a warm instance remains in the pool after execution. 4. **Adjust `PreWarmThreshold`**: Lower the threshold if traffic patterns are predictable.
Warm pool consumes excessive memory even when idle	Too many instances are kept warm (`MaxWarmInstances` is too high), or instances aren't being released after use (`ReleaseInstance` isn't called). This wastes resources and can lead to host memory pressure.	1. **Monitor pool size**: Track `ContainerCount` gauge over time and set alerts for high idle counts. 2. **Implement memory-based eviction**: Add logic to `cleanupExpired` that removes instances when system memory pressure is high. 3. **Adjust per-function limits**: Set `MaxWarmInstances` based on actual usage patterns—most functions need only 1-2 warm instances. 4. **Add LRU eviction**: When the pool is full, evict the least recently used instance rather than rejecting new ones.
Function state persists between invocations (unexpected side effects)	The container isn't being properly reset between uses. The `CleanupInstance` method might not be clearing the `/tmp` directory or environment variables from previous invocations.	1. **Verify cleanup hooks**: Ensure `RuntimeCleaner.Clean` is called between invocations and resets the filesystem state. 2. **Check filesystem isolation**: Each invocation should get a fresh overlay filesystem layer; ensure `Binds` includes a unique `/tmp` mount. 3. **Test with a counter**: Write a function that increments a counter in `/tmp/count.txt`—it should reset to 1 on each cold start and warm reuse. 4. **Use read-only root filesystem**: Mount the function code as read-only and only allow writes to `/tmp`.
Snapshot/restore (CRIU) fails with "pre-dump not found" error	The container state is too complex to checkpoint (e.g., open network connections, pending signals), or CRIU isn't properly configured for the container runtime. This prevents fast restoration from snapshots.	1. **Simplify container state**: Ensure containers are checkpointed at a consistent point (e.g., after initialization but before any request). 2. **Check CRIU version**: Use a recent version (≥3.15) with full container support. 3. **Pre-dump memory**: Use iterative memory dumping to reduce pause times during checkpoint. 4. **Fallback to traditional start**: If snapshot fails after retries, fall back to regular container creation and log the error for investigation.
Predictive warming creates instances that are never used	The prediction algorithm is too aggressive or based on noisy metrics. This wastes resources and can cause scale-up/down oscillations.	1. **Tune prediction parameters**: Adjust `PreWarmThreshold` based on actual hit rate—monitor how many pre-warmed instances are actually used. 2. **Add confidence scoring**: Only pre-warm when the prediction confidence exceeds a threshold. 3. **Correlate with external metrics**: Use request queue depth or upstream event sources (like message queue backlog) as stronger signals. 4. **Implement graceful degradation**: If predictive warming accuracy is low, disable it and rely on reactive warming only.

Request Routing and Gateway Issues

These symptoms affect request handling, load balancing, and queuing behavior.

Symptom	Likely Cause	Steps to Fix
Requests hang indefinitely without timeout	The gateway's timeout chain is broken—`GatewayConfig.DefaultTimeout` might not be propagating to the `Router.Route` context, or the `forwardRequest` isn't respecting the deadline. This causes client connections to hang.	1. **Check timeout propagation**: Use `withTimeout` at each layer (gateway → router → instance) and ensure deadlines are decreasing. 2. **Verify context cancellation**: When a request times out, ensure all downstream operations (container execution, network calls) are cancelled. 3. **Add request-scoped timeouts**: The `InvocationRequest.Deadline` should be set by the gateway and respected throughout. 4. **Test with slow functions**: Create a function that sleeps longer than the timeout and verify it's properly terminated.
Load balancing sends all requests to one instance	The load balancing algorithm (likely in `selectInstance`) is stuck on a single instance due to sticky sessions or a bug in least-connections counting. This overloads one instance while others sit idle.	1. **Check `routedInstance.activeReqs`**: Ensure the atomic counter is being incremented/decremented correctly in `forwardRequest`. 2. **Disable sticky routing**: If `RouterConfig.StickySessionTTL` > 0, set it to 0 to test if the issue persists. 3. **Verify instance health**: Unhealthy instances might be excluded, leaving only one healthy instance. Check `circuitBreaker` state. 4. **Test with concurrent requests**: Send multiple parallel requests and verify they distribute across all warm instances.
Request queue grows without bound, causing memory exhaustion	The `RequestQueue` isn't bounded properly (`maxSize` is too large or not enforced), or the dequeue rate is slower than enqueue rate due to stuck instances. This can lead to the gateway being OOM-killed.	1. **Check queue bounds**: Verify `NewRequestQueue` is called with a reasonable `maxSize` (e.g., 1000 per function). 2. **Implement load shedding**: When queue is full, reject new requests with 429 (Too Many Requests) instead of queuing. 3. **Monitor dequeuing rate**: If `Dequeue` is slow, check for lock contention in the queue or blocked workers. 4. **Add queue timeouts**: Use `RemoveExpired` to clean up requests that have waited too long.
Circuit breaker trips unnecessarily, causing healthy instances to be avoided	The `CircuitBreakerThreshold` is too low or the timeout (`CircuitBreakerTimeout`) is too short for normal function execution. This causes temporary slowness to be interpreted as failure.	1. **Adjust breaker parameters**: Increase the threshold (e.g., from 5 to 10 failures) and lengthen the timeout to match function SLA. 2. **Distinguish error types**: Only count network errors or timeouts toward the breaker, not application errors (like 400 Bad Request). 3. **Add half-open state**: After the timeout, allow a trial request before fully closing the breaker again. 4. **Monitor breaker metrics**: Track how often breakers trip and for which functions—adjust per-function if needed.
Gateway returns 502 Bad Gateway but instances appear healthy	The network between gateway and instances is failing, or the instance health check (`IsActive`) is incorrectly reporting healthy when the container is actually dead. This can happen if the container died but the `FunctionInstance` state wasn't updated.	1. **Check instance connectivity**: Use `net.Dial` from the gateway to the instance's `Host:Port` to verify TCP connectivity. 2. **Implement active health checks**: The `Router` should periodically ping instances (HTTP GET /health) and mark unhealthy ones. 3. **Watch for state desync**: Ensure lifecycle events (`EventHealthCheckFailed`, `EventForceTerminate`) update the `FunctionInstance.State` promptly. 4. **Add retry with different instance**: On 502, the gateway should retry the request with a different instance before failing.

Auto-Scaling and Resource Management Issues

These symptoms involve incorrect scaling decisions, instance churn, or resource waste.

Symptom	Likely Cause	Steps to Fix
Scaling oscillates rapidly (thrashing) — instances constantly created and destroyed	The scaling thresholds (`ScaleUpThreshold` , `ScaleDownThreshold`) are too close together, or the cooldown periods (`ScaleUpCooldown` , `ScaleDownCooldown`) are too short. This causes the system to overreact to small load fluctuations.	1. **Add hysteresis**: Ensure scale-up threshold (e.g., 70% concurrency) is significantly higher than scale-down threshold (e.g., 30%). 2. **Increase cooldowns**: Set cooldowns to at least 30-60 seconds to prevent rapid successive decisions. 3. **Smooth metrics**: Use moving averages for concurrency/RPS instead of instantaneous values. 4. **Implement scaling stabilization**: Keep track of recent scaling actions and suppress opposite-direction actions for a stabilization window.
Scale-to-zero removes instances too aggressively, causing cold starts on every request	The `IdleTimeout` is too short for the function's invocation pattern. Intermittent workloads (e.g., cron jobs every 5 minutes) need longer idle timeouts than the interval between invocations.	1. **Analyze invocation patterns**: Check metrics for the time between invocations—set `IdleTimeout` to at least 2-3x the typical interval. 2. **Add predictive keep-alive**: If historical data shows regular intervals, keep at least one instance warm for predictable workloads. 3. **Implement per-function tuning**: Some functions may need longer idle timeouts than others. 4. **Consider minimum instances**: For latency-sensitive functions, set `MinInstances` to 1 instead of relying on scale-to-zero.
Scale-up is too slow during traffic spikes, causing queue buildup	The scaling decision loop (`EvaluateScaling`) runs too infrequently, or the metrics window (`MetricsWindow`) is too long, causing delayed detection of traffic increases.	1. **Reduce evaluation interval**: The `Scaler.ticker` should run at least every 5 seconds during high traffic. 2. **Use faster metrics**: Combine short-term (5s) and long-term (60s) metrics to react quickly while avoiding noise. 3. **Implement predictive scaling**: If traffic patterns are predictable (e.g., daily peaks), scale up proactively. 4. **Add queue-based scaling**: Scale up when request queue depth exceeds a threshold, not just based on concurrency.
Scale-down doesn't happen even when instances are idle for hours	The scale-down logic requires instances to be idle *and* below threshold for the entire cooldown period, or there's a bug in `IdleDuration` calculation (`LastUsedAt` isn't updated properly).	1. **Verify `LastUsedAt` updates**: Ensure `EventRequestCompleted` updates the instance's `LastUsedAt` timestamp. 2. **Check idle detection**: The scaler should consider both low concurrency AND idle time before scaling down. 3. **Test with synthetic load**: Create a burst of traffic, then let the system idle and verify scale-down occurs after `IdleTimeout` . 4. **Add forced scale-down**: If instances have been idle for an extended period (e.g., 24 hours), force termination regardless of thresholds.
Provisioned concurrency instances aren't kept warm	The warm pool manager isn't aware of provisioned concurrency requirements, or the instances are being recycled due to TTL expiration. This defeats the purpose of provisioned concurrency.	1. **Integrate with warm pool**: The `Scaler` should call `PreWarm` for provisioned concurrency instances and ensure they're excluded from TTL-based cleanup. 2. **Mark provisioned instances**: Add a label to instances created for provisioned concurrency so the pool manager treats them specially. 3. **Monitor warm count**: Alert if the number of warm instances for a function with provisioned concurrency drops below the minimum. 4. **Implement health checks**: Provisioned instances should have active health checks and be recreated if unhealthy.

Cross-Component Integration Issues

These symptoms involve interactions between multiple components and are often the trickiest to debug.

Symptom	Likely Cause	Steps to Fix
Lifecycle events are lost or processed out of order	The `EventDispatcher` has subscribers with small buffers, or events are being published from multiple goroutines without proper ordering guarantees. This can cause state machine corruption.	1. **Increase buffer sizes**: Use buffered channels sized for expected event volume (e.g., 1000). 2. **Sequence events**: Add a monotonically increasing sequence number to `InstanceEvent` and have subscribers detect gaps. 3. **Use a single publisher**: Ensure all events go through a single goroutine that sequences them before publishing. 4. **Add event persistence**: For critical events, write them to a WAL before processing so they can be replayed after crashes.
State machine gets stuck in `InstanceStateDraining` indefinitely	The drain timeout is too long, or the instance still has active requests that aren't completing. This blocks scale-down and wastes resources.	1. **Check active request count**: Verify `routedInstance.activeReqs` reaches zero during draining. 2. **Enforce drain timeout**: The `FlowCoordinator` should force-terminate instances after `DrainTimeout`. 3. **Verify request completion**: Ensure `EventRequestCompleted` is emitted even for failed requests. 4. **Add watchdog timer**: If an instance stays draining beyond timeout, emit `EventForceTerminate`.
Memory leak in gateway or controller over time	Goroutines aren't being cleaned up after requests complete, or objects are retained in global maps (like `Router.functions`) without cleanup. This is common with improper context usage.	1. **Profile heap usage**: Use `pprof` to see which types are accumulating. 2. **Check for goroutine leaks**: Monitor goroutine count over time; use `net/http/pprof` to see stack traces. 3. **Clean up stale entries**: Remove function entries from the router when functions are deleted. 4. **Use weak references**: For caches, use `sync.Map` with time-based expiration.
Metrics show unrealistic values (e.g., negative durations)	Race conditions in metric updates, or timestamps being recorded in different time sources (e.g., `time.Now()` vs `time.Since()`). This makes monitoring unreliable.	1. **Use atomic operations**: For counters and gauges updated concurrently, use `atomic` package or Prometheus's built-in atomicity. 2. **Standardize time source**: Use `time.Now()` at the start of operations and calculate durations from that single reference. 3. **Validate metrics**: Add sanity checks (e.g., duration ≥ 0) before recording. 4. **Test with race detector**: Run tests with `-race` flag to catch concurrent updates.

Implementation Guidance

> **Debugging Philosophy**: The best debugging strategy is **observability by design**—instrument first, debug second. Add structured logs, metrics, and traces during initial implementation rather than as an afterthought.

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Logging	Structured JSON logging with `log/slog` (Go 1.21+) or `zerolog`	Distributed tracing with OpenTelemetry + Jaeger
Metrics	Prometheus client library with built-in `/metrics` endpoint	Prometheus + Grafana with custom dashboards per component
Tracing	Request IDs propagated via HTTP headers	Full OpenTelemetry instrumentation with span context
Profiling	`net/http/pprof` endpoints for CPU/memory profiles	Continuous profiling with Pyroscope or Parca
Debug Tools	`delve` debugger for Go, `docker inspect` for containers	`kubectl debug` for live container inspection

B. Recommended Debugging Infrastructure

Create a dedicated `internal/observability/` directory for shared debugging utilities:

```
project-root/
  internal/observability/
    logging.go          ← Structured logger setup
    metrics.go          ← Prometheus metric registration helpers
    tracing.go          ← OpenTelemetry tracer provider
    middleware.go       ← HTTP middleware for request ID, logging
    debug_endpoints.go  ← pprof, health check endpoints
  cmd/debug-tools/      ← Standalone debugging utilities
    container-inspector/ ← Tool to examine container state
    trace-replayer/     ← Replay captured invocation traces
```

C. Complete Debugging Helper: Request ID Propagation

Here's a complete implementation for request ID propagation across components—essential for tracing a single invocation through the system:

GO

```go
// internal/observability/middleware.go

package observability


import (

    "context"

    "net/http"

    "github.com/google/uuid"

)


type contextKey string


const requestIDKey contextKey = "request_id"


// RequestIDMiddleware adds a unique request ID to all incoming requests

func RequestIDMiddleware(next http.Handler) http.Handler {

    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {

        // Check for existing request ID header (for trace continuity)

        requestID := r.Header.Get("X-Request-ID")

        if requestID == "" {

            requestID = uuid.New().String()

        }



        // Add to response headers for client debugging

        w.Header().Set("X-Request-ID", requestID)



        // Store in context for downstream components

        ctx := context.WithValue(r.Context(), requestIDKey, requestID)

        next.ServeHTTP(w, r.WithContext(ctx))

    })

}


// GetRequestID extracts the request ID from context

func GetRequestID(ctx context.Context) string {

    if id, ok := ctx.Value(requestIDKey).(string); ok {

        return id

    }

    return "unknown"

}


// LoggingMiddleware adds structured logging with request ID

func LoggingMiddleware(logger *slog.Logger, next http.Handler) http.Handler {

    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {

        start := time.Now()
```

```
        requestID := GetRequestID(r.Context())

        // Wrap response writer to capture status code
        rw := &responseWriter{ResponseWriter: w, statusCode: http.StatusOK}

        // Log request start
        logger.InfoContext(r.Context(), "request started",
            "method", r.Method,
            "path", r.URL.Path,
            "request_id", requestID,
        )

        next.ServeHTTP(rw, r)

        // Log request completion
        logger.InfoContext(r.Context(), "request completed",
            "method", r.Method,
            "path", r.URL.Path,
            "status", rw.statusCode,
            "duration_ms", time.Since(start).Milliseconds(),
            "request_id", requestID,
        )
    })
}

type responseWriter struct {
    http.ResponseWriter
    statusCode int
}

func (rw *responseWriter) WriteHeader(statusCode int) {
    rw.statusCode = statusCode
    rw.ResponseWriter.WriteHeader(statusCode)
}
```

D. Core Debugging Skeleton: Diagnostic Endpoint

Add a diagnostic endpoint to the gateway that reveals internal state—crucial for debugging routing and scaling issues:

```go
// internal/gateway/debug.go

package gateway


import (

    "encoding/json"

    "net/http"

    "sync/atomic"

)


// DebugHandler exposes internal state for debugging

type DebugHandler struct {

    router *Router

    warmPool pool.WarmPoolManager

    scaler *scaler.Scaler

}


// ServeHTTP handles debug requests

func (h *DebugHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {

    switch r.URL.Query().Get("view") {

    case "routing":

        h.handleRoutingDebug(w, r)

    case "warmpool":

        h.handleWarmPoolDebug(w, r)

    case "scaling":

        h.handleScalingDebug(w, r)

    default:

        h.handleOverview(w, r)

    }

}


// handleRoutingDebug shows current routing table state

func (h *DebugHandler) handleRoutingDebug(w http.ResponseWriter, r *http.Request) {

    h.router.mu.RLock()

    defer h.router.mu.RUnlock()



    type instanceInfo struct {

        ID         string    `json:"id"`

        State      string    `json:"state"`

        Host       string    `json:"host"`

        Port       int       `json:"port"`

        ActiveReqs int32     `json:"active_reqs"`

        LastUsed   time.Time `json:"last_used"`
```

```
        Healthy    bool      `json:"healthy"`
    }

    type functionInfo struct {
        Name          string        `json:"name"`
        Instances     []instanceInfo `json:"instances"`
        ConcurrencyLimit int        `json:"concurrency_limit"`
        ActiveCount   int32         `json:"active_count"`
    }

    functions := make([]functionInfo, 0, len(h.router.functions))
    for name, rt := range h.router.functions {
        instances := make([]instanceInfo, 0, len(rt.instances))
        for _, ri := range rt.instances {
            instances = append(instances, instanceInfo{
                ID:         ri.instance.ID,
                State:      string(ri.instance.State),
                Host:       ri.instance.Host,
                Port:       ri.instance.Port,
                ActiveReqs: atomic.LoadInt32(&ri.activeReqs),
                LastUsed:   ri.lastUsed,
                Healthy:    ri.healthy,
            })
        }
        functions = append(functions, functionInfo{
            Name:            name,
            Instances:       instances,
            ConcurrencyLimit: int(rt.concurrencyLimit),
            ActiveCount:     atomic.LoadInt32(&rt.activeCount),
        })
    }

    w.Header().Set("Content-Type", "application/json")
    json.NewEncoder(w).Encode(map[string]interface{}{
        "timestamp": time.Now().Format(time.RFC3339),
        "functions": functions,
    })
}

// TODO: Implement handleWarmPoolDebug to show warm pool contents
// TODO: Implement handleScalingDebug to show scaling decisions and metrics
```

```
// TODO: Implement handleOverview to show system health summary
```

E. Language-Specific Debugging Hints for Go

1. **Memory Profiling**: Run `go tool pprof http://localhost:6060/debug/pprof/heap` to analyze memory usage. Use `-base` flag to compare two profiles.
2. **Goroutine Leaks**: Check `/debug/pprof/goroutine?debug=2` for full stack traces. Look for goroutines stuck in channel operations without timeouts.
3. **Race Detection**: Always run tests with `go test -race ./...`. For production debugging, build with `-race` flag (2-10x slowdown) during staging.
4. **Context Timeout Chains**: Use `context.WithTimeout` at each layer and ensure child contexts are cancelled when parent times out. Add logging when contexts are cancelled to trace timeout propagation.
5. **Container Debugging**: When a container fails, capture its logs before removal. Store logs in a ring buffer keyed by container ID, accessible via debug API.

F. Debugging Workflow Checklist

When encountering a bug, follow this systematic workflow:

1. **Reproduce the Issue**
   - Can you reproduce it consistently? If not, add more logging and try again.
   - What are the minimal conditions (function code, load pattern, configuration)?
2. **Check the Obvious First**
   - Are all services running? (`docker ps`, `systemctl status`)
   - Are there error logs in any component? (Check `journalctl` or component logs)
   - Is the system under resource pressure? (`htop`, `df -h`, `docker stats`)
3. **Isolate the Component**
   - Does the issue happen during packaging, routing, execution, or scaling?
   - Can you bypass components (e.g., invoke container directly vs through gateway)?
   - Use the debug endpoints (`/debug/routing`, `/debug/warmpool`) to inspect state.
4. **Add Targeted Instrumentation**
   - Increase log level for the suspected component.
   - Add temporary metrics or traces around the problematic code path.
   - Use conditional breakpoints if debugging locally.
5. **Analyze the Evidence**
   - Collect logs, metrics, and profiles from the time of the issue.
   - Look for patterns (e.g., "always fails after 100 invocations", "only when memory > 80%").
   - Correlate events across components using request IDs.
6. **Form and Test Hypothesis**
   - Based on evidence, what's the most likely root cause?
   - Can you write a test that reproduces the issue?
   - Fix the issue and verify with the same reproduction steps.

G. Common Debugging Command Cheat Sheet

```bash
# View container logs for a specific function instance
docker logs $(docker ps -qf "label=function.name=myfunc")

# Inspect container resource limits
docker inspect $(docker ps -qf "name=worker") --format='{{.HostConfig.Memory}} {{.HostConfig.CpuQuota}}'

# Check cgroup memory usage for a container
cat /sys/fs/cgroup/memory/docker/<container-id>/memory.usage_in_bytes

# Profile Go service with pprof
go tool pprof -http=:8080 http://localhost:6060/debug/pprof/profile?seconds=30

# Trace HTTP requests through the system
curl -H "X-Request-ID: debug-123" http://gateway/invoke/myfunc

# Monitor scaling decisions in real-time
watch -n 1 'curl -s http://gateway/debug?view=scaling | jq'

# Force garbage collection and view memory stats
curl http://localhost:6060/debug/pprof/heap?gc=1

# Check for goroutine leaks
curl -s http://localhost:6060/debug/pprof/goroutine?debug=1 | grep -c "goroutine"
```

Remember: **The most important debugging tool is a reproducible test case.** When you encounter a bug, immediately write a test that reproduces it before fixing it. This prevents regression and helps understand the root cause.

Future Extensions

> **Milestone(s):** This forward-looking section describes enhancements that build upon the foundational architecture established in Milestones 1-5. It illustrates how the current design's modularity and clear interfaces naturally accommodate evolving requirements.

The serverless function runtime has been designed with **extensibility as a first-class concern**. Its component-based architecture, well-defined interfaces, and event-driven communication patterns create a foundation that can gracefully accommodate new capabilities without requiring significant refactoring. This section explores several potential enhancements that could be built upon the current system, explaining how each aligns with existing components and what modifications would be required.

Custom Domains and API Gateway Features

Mental Model: The Hotel Concierge Service Imagine upgrading from a simple reception desk (the current HTTP gateway) to a full-service hotel concierge. Instead of just directing guests to their rooms (functions), the concierge can handle complex requests: "Route `api.example.com/v1/orders` to the `order-processor` function, but send `webhooks.example.com` to the `webhook-handler` function, and apply rate limiting of 100 requests per minute for the `public-api` function." This enhanced routing layer transforms the runtime from a simple function executor into a full-fledged API management platform.

Extension Description: Currently, the `Gateway` routes all requests through a single endpoint pattern (`/invoke/{functionName}`). A custom domains extension would allow users to map their own domain names to specific functions or groups of functions, with advanced routing rules, authentication, and rate limiting.

How Current Design Accommodates It:

- The `Gateway` component already provides a pluggable middleware chain via the `applyMiddleware` method, making it straightforward to add authentication, rate limiting, and request transformation logic.
- The `Router` already performs function-based routing; extending it to support host/path-based routing would primarily involve enhancing the `functionRoutingTable` to support multiple lookup keys (e.g., hostname + path prefix).
- The `InvocationRequest` type already includes a `Source` field that could be enriched with original host/path information for routing decisions.

Required Changes:

1. **Enhanced Routing Table:** Extend the `functionRoutingTable` to support mapping from `(host, path)` tuples to a target function, with priority matching for path prefixes.
2. **Domain Configuration Store:** Add a new configuration component to manage domain-to-function mappings, likely persisted in the same storage backend used for function artifacts.
3. **Gateway Middleware:** Implement new middleware for TLS termination (if handling HTTPS), JWT validation, API key checking, and rate limiting per API key or IP address.
4. **Public API Endpoint:** Add a control plane API for users to configure custom domains and routing rules.

Example Routing Configuration Table:

Host	Path Pattern	Target Function	Rate Limit	Authentication
`api.company.com`	`/v1/orders/*`	`order-processor`	1000 RPM	JWT
`webhooks.company.com`	`/stripe`	`stripe-webhook`	None	HMAC Signature
`(default)`	`/admin/*`	`admin-backend`	100 RPM	API Key

Function Composition and Workflows

Mental Model: The Assembly Line Think of moving from a single, isolated workshop (individual function) to a coordinated assembly line. A raw material (request) enters the line, passes through station A (function A) for initial processing, then moves to station B (function B) for enrichment, and finally to station C (function C) for packaging the response. The assembly line controller manages the flow, handles errors at any station, and ensures the final product is delivered. This enables building complex applications by composing small, reusable functions.

Extension Description: Function composition allows users to define workflows where the output of one function becomes the input to another, either sequentially or in parallel. This enables building complex serverless applications without requiring a separate orchestration service.

How Current Design Accommodates It:

- The existing `InvocationRequest` and `InvocationResponse` structures already encapsulate payloads and metadata, providing a natural format for passing data between functions.
- The `Gateway` and `Router` components already manage request routing; they could be extended to support internal function-to-function calls without leaving the cluster (reducing latency).
- The event-driven architecture with the `EventDispatcher` provides a foundation for choreographing workflows, where completion of one function triggers the next.

Required Changes:

1. **Workflow Definition Language:** Introduce a new `WorkflowDefinition` type that describes the sequence, parallel branches, error handling, and retry logic.
2. **Workflow Engine:** Create a new `WorkflowExecutor` component that interprets workflow definitions, manages state (e.g., which step is next), and emits events for each step's start/completion.
3. **Internal Service Discovery:** Enhance the `Router` to support routing to a local function instance without going through the public gateway, perhaps via a dedicated internal hostname or port.
4. **State Persistence:** For long-running workflows, introduce a durable state store (e.g., Redis or a database) to track workflow execution progress, allowing recovery after failures.

Example Workflow Definition Structure:

Field	Type	Description
`Name`	`string`	Unique workflow identifier
`Version`	`string`	Immutable version tag
`StartAt`	`string`	Name of the first step
`Steps`	`map[string]WorkflowStep`	Map of step name to step definition
`Timeout`	`int`	Maximum total execution time in seconds

WorkflowStep Structure:

Field	Type	Description
`Type`	`string`	`"function"`, `"parallel"`, `"choice"`
`FunctionName`	`string`	Target function (for `"function"` type)
`Next`	`string`	Name of next step on success
`Catch`	`[]ErrorHandler`	Array of error handlers for this step
`Retry`	`[]RetryPolicy`	Array of retry policies for transient failures

Advanced Observability and Distributed Tracing

Mental Model: The Air Traffic Control Radar Imagine upgrading from simple flight logs (current metrics) to a comprehensive air traffic control radar system. You can now see not only each plane's (function's) current status but also its entire flight path (request journey) through different airspaces (components), with detailed telemetry on altitude (resource usage), speed (latency), and any turbulence (errors). This end-to-end visibility is crucial for debugging complex interactions in a distributed system.

Extension Description: While the current design includes basic metrics via Prometheus (`Metrics` type), advanced observability would add distributed tracing (e.g., OpenTelemetry), structured logging with correlation IDs, and a real-time dashboard for visualizing function dependencies, invocation chains, and performance bottlenecks.

How Current Design Accommodates It:

- The `InvocationRequest` already includes a `RequestID` field, which can be propagated through all components to correlate logs and traces.
- The `EventDispatcher` already emits lifecycle events (`InstanceEvent`); these events could be automatically enriched with trace spans and exported to an observability backend.
- The middleware architecture in the `Gateway` makes it easy to inject and extract trace context headers (e.g., W3C Trace Context).

Required Changes:

1. **Tracing Integration:** Integrate an OpenTelemetry SDK to create spans for key operations: gateway request handling, routing, container creation, function execution, and scaling decisions.
2. **Context Propagation:** Ensure the trace context is passed through the `InvocationRequest` (e.g., via a `TraceContext map[string]string` field) and used when making internal HTTP calls between components.
3. **Enhanced Logging:** Replace simple log statements with structured logging using a library like `slog` (Go 1.21+), ensuring all logs include the request ID, function name, and trace ID.
4. **Observability Backend:** Deploy a tracing backend (e.g., Jaeger or Tempo) and a log aggregation system (e.g., Loki) to store and query traces and logs.

Example Trace Context Addition to `InvocationRequest`:

```go
type InvocationRequest struct {
    // ... existing fields ...
    TraceContext map[string]string `json:"traceContext,omitempty"` // W3C Trace Context headers
}
```

Observability Event Flow:

1. Gateway receives request → creates root span, extracts/injects trace context.
2. Router selects instance → adds span for routing logic.
3. Warm pool provides instance → adds span for pool acquisition.
4. Function executes → adds span for execution duration.
5. All spans are linked via trace ID and exported to collector.

Stateful Functions and Ephemeral Storage

Mental Model: The Hotel Room with a Safe Currently, each function invocation gets a completely fresh "pop-up shop" with no memory of previous customers. Imagine if each shop could have a small, private safe (`ephemeral storage`) that persists only for the lifetime of that specific shop instance. Multiple visits by the same customer (requests routed to the same instance) could access and update the safe's contents, enabling caching or session state, but when the shop closes (instance terminates), the safe is destroyed. This provides limited, instance-local state without the complexity of external databases.

Extension Description: While serverless functions are ideally stateless, some workloads benefit from temporary, instance-local storage that persists across invocations on the same instance (e.g., caching fetched data, accumulating metrics, or maintaining WebSocket connections). This extension would provide a managed, filesystem-based cache that lives alongside the function instance and is automatically cleaned up when the instance is terminated.

How Current Design Accommodates It:

- The `FunctionInstance` already has a lifecycle and identity (`ID`). The `ContainerManager` could attach a volume or directory that persists as long as the container lives.
- The `WarmPoolManager` already manages instance reuse; it could preserve the ephemeral storage when returning an instance to the pool.
- The `RuntimeCleaner` interface already defines a `Clean` method, which could be extended to optionally preserve (or securely wipe) the ephemeral storage directory.

Required Changes:

1. **Ephemeral Storage Configuration:** Add a field to `FunctionDefinition` to request ephemeral storage (size, mount path).
2. **Container Volume Attachment:** Extend `CreateContainerRequest` to include volume mounts for ephemeral storage, using a unique directory per `FunctionInstance.ID` .
3. **Lifecycle Management:** Modify the `WarmPoolManager` to not delete the ephemeral storage when releasing an instance back to the pool (only on full termination).
4. **Security Hardening:** Ensure the ephemeral storage is isolated per function and instance (using filesystem permissions or separate volumes) to prevent cross-tenant access.

Example Addition to `FunctionDefinition` :

Field	Type	Description
`EphemeralStorageMB`	`int`	Size in MB of instance-local persistent storage (0 for none)
`EphemeralStoragePath`	`string`	Mount path inside container (default `/cache`)

GPU and Specialized Hardware Support

Mental Model: The Specialty Kitchen Our current "pop-up shops" are equipped with standard kitchen appliances (CPU, memory). Some recipes (workloads) require specialty equipment: a blast chiller for rapid cooling (GPU for matrix math), a sous-vide precision cooker (FPGA for cryptographic operations), or a commercial dough sheeter (TPU for machine learning). This extension allows users to request shops with specific hardware accelerators for compute-intensive tasks.

Extension Description: Extend the runtime to schedule functions on worker nodes with specialized hardware (GPUs, FPGAs, TPUs) and expose that hardware to the function container. This opens up serverless to machine learning inference, video encoding, scientific computing, and other accelerated workloads.

How Current Design Accommodates It:

- The `ContainerManager` interface already abstracts container creation; implementations (like `DockerManager`) can be extended to pass device mappings (`--device` flag in Docker).
- The `Executor` already handles resource limits (CPU, memory); it could be extended to request and track specialized hardware units.
- The `Scaler` would need awareness of heterogeneous worker nodes (some with GPU, some without) to make appropriate scaling decisions.

Required Changes:

1. **Hardware Resource Model:** Extend `FunctionDefinition` with optional fields like `GPUCount` , `GPUType` , `FPGA` , etc.
2. **Node Labeling and Scheduling:** Implement a labeling system for worker nodes (e.g., `hardware:gpu-p100`) and modify the `Executor` to select a node with matching capabilities when creating a container.
3. **Container Device Mapping:** Update `CreateContainerRequest` to include device requests, which the `ContainerManager` implementation translates to runtime-specific flags.
4. **Cost and Billing:** Track usage of specialized hardware separately for metering and billing purposes.

Example Addition to `CreateContainerRequest` :

Field	Type	Description
`Devices`	`[]DeviceMapping`	List of host devices to map into container
`DeviceRequests`	`[]DeviceRequest`	Docker-specific device requests (for GPU)

DeviceMapping Structure:

Field	Type	Description
`HostPath`	`string`	Path on host
`ContainerPath`	`string`	Path in container
`Permissions`	`string`	`"rwm"` (read, write, mknod)

Multi-Cloud and Hybrid Deployment

Mental Model: The Franchise Model Our current runtime operates in a single data center (cloud region). Imagine expanding to a franchise model: identical shops (function instances) can be launched in multiple locations (clouds or on-premises), with a central management system that routes customers to the nearest location or the one with lowest latency. This provides resilience against regional outages and compliance with data sovereignty requirements.

Extension Description: Deploy the serverless runtime control plane once, but have it manage worker pools across multiple cloud providers (AWS, GCP, Azure) and/or on-premises data centers. Functions can be deployed globally, with invocations routed to the nearest available instance.

How Current Design Accommodates It:

- The architecture already separates the **control plane** (Gateway, Controller, Scaler) from the **data plane** (Worker Pool). This separation allows the control plane to manage multiple remote worker pools.
- The `ContainerManager` interface abstracts the underlying infrastructure; different implementations could target different clouds (e.g., `AWSLambdaManager`, `KubernetesManager`).
- The `Router` could be enhanced with geo-aware routing based on the source IP of the request.

Required Changes:

1. **Multi-Cluster Management:** Extend the `Scaler` and `WarmPoolManager` to manage pools of instances across multiple clusters, each with its own `ContainerManager`.
2. **Service Discovery Across Clouds:** Implement a global service discovery mechanism (e.g., using a distributed key-value store like etcd) to track available instances across regions.
3. **Latency-Based Routing:** Enhance the `Router` with latency measurements or geo-IP databases to select the optimal region for each request.
4. **Cross-Cloud Networking:** Establish secure network connectivity (VPN or cloud interconnect) between control plane and worker pools in different clouds.

Example Multi-Cluster Configuration:

Cluster Name	Provider	Region	ContainerManager Type	Endpoint
`us-east`	AWS	us-east-1	`DockerManager`	`https://worker-us-east.internal`
`eu-central`	GCP	europe-west3	`KubernetesManager`	`https://worker-eu.internal`
`on-prem`	On-prem	datacenter-1	`DockerManager`	`https://worker-onprem.internal`

Event-Driven Architecture with Multiple Sources

Mental Model: The Universal Adapter Hub Currently, functions are primarily triggered by HTTP requests. Imagine adding a universal adapter hub that can connect to any event source: message queues (Kafka, RabbitMQ), cloud storage (S3 bucket notifications), database change streams (MongoDB change streams, PostgreSQL LISTEN/NOTIFY), or cron schedules. The hub converts these diverse events into a standard `InvocationRequest` and feeds them into the existing gateway or a dedicated async invoker.

Extension Description: Expand the event sources beyond HTTP to include message queues, streaming platforms, cloud storage events, and scheduled (cron) invocations. This transforms the runtime into a general-purpose event-driven compute platform.

How Current Design Accommodates It:

- The `InvocationRequest` type is already source-agnostic, with a `Source` field indicating origin.
- The asynchronous invocation mode (queued request) already decouples event receipt from function execution.
- The `EventDispatcher` can be used to propagate events from new sources to internal components.

Required Changes:

1. **Event Source Connectors:** Develop a set of pluggable connectors (`KafkaConnector`, `S3EventsConnector`, `CronScheduler`) that poll or listen for events and convert them to `InvocationRequest`s.
2. **Event Router:** Create a new `EventRouter` component that receives events from all connectors, applies optional filtering/transformation, and forwards them to the appropriate function (via the existing `Gateway` or a new async path).
3. **Enhanced Async Invoker:** Extend the asynchronous invocation pathway to support larger payloads (by storing them in object storage) and guaranteed delivery with retries (using a durable queue like Redis Streams or Apache Pulsar).

Event Source Connector Interface:

Method	Parameters	Returns	Description
`Start`	`ctx context.Context, eventHandler func(InvocationRequest) error`	`error`	Starts listening for events and calls handler for each
`Stop`	`ctx context.Context`	`error`	Stops the connector gracefully
`Health`	`ctx context.Context`	`error`	Returns health status of the connection to event source

Design Principles for Extensibility

Throughout these extensions, several core design principles embedded in the current architecture prove invaluable:

1. **Interface-Based Abstraction:** Key components like `ContainerManager`, `WarmPoolManager`, and `EventDispatcher` are defined as interfaces, allowing multiple implementations without changing dependent code.
2. **Event-Driven Communication:** The lifecycle event system (`InstanceEvent`) creates a loosely coupled architecture where new components can subscribe to relevant events without modifying event publishers.
3. **Modular Configuration:** The use of configuration structs (`GatewayConfig`, `ScalingConfig`, `PoolConfig`) makes it easy to add new knobs for tuning extended features.
4. **Clear Data Flow:** The well-defined data structures (`FunctionDefinition`, `InvocationRequest`, `FunctionInstance`) serve as a consistent "language" across components, making it natural to add new fields for extended capabilities.
5. **Layered Architecture:** Separation between control plane (orchestration) and data plane (execution) allows each to evolve independently and scale differently.

By adhering to these principles, the serverless function runtime maintains its agility—ready to evolve as new requirements emerge from users and the ever-changing landscape of cloud computing.

Implementation Guidance

Technology Recommendations Table:

Extension	Simple Option	Advanced Option
Custom Domains	Nginx proxy in front of Gateway with manual config	Envoy Proxy with dynamic configuration via xDS API
Function Composition	Simple sequential chaining in Gateway middleware	Apache Airflow or Temporal integration for complex DAGs
Distributed Tracing	OpenTelemetry SDK with stdout exporter	Jaeger or Grafana Tempo backend with sampling and aggregation
Ephemeral Storage	Host-mounted volumes with per-instance directories	Network-attached storage (e.g., EFS) with faster cleanup
GPU Support	Docker `--gpus all` flag with NVIDIA runtime	Kubernetes Device Plugins with resource quotas
Multi-Cloud	Separate deployments per cloud with manual failover	HashiCorp Consul for service mesh across clouds
Event Sources	Cron scheduler built into Controller	Apache Camel or Spring Cloud Stream for connector framework

Recommended File/Module Structure:

```
project-root/
  cmd/
    server/                # Main control plane server
    worker/                # Optional separate worker agent
  internal/
    extensions/            # New directory for extension implementations
      apigateway/          # Custom domains and API Gateway features
        router_ext.go      # Enhanced routing table
        middleware/        # Auth, rate limiting middleware
      workflows/           # Function composition
        workflow_def.go    # WorkflowDefinition type
        executor.go        # WorkflowExecutor
      observability/       # Advanced tracing
        tracing.go         # OpenTelemetry setup
        correlated_logger.go # Structured logging
      storage/             # Ephemeral storage
        ephemeral_manager.go # Manages instance-local volumes
      hardware/            # GPU support
        device_manager.go  # Detects and allocates devices
      multicloud/          # Multi-cloud deployment
        cluster_manager.go # Manages multiple worker clusters
      eventsources/        # Event-driven sources
        connectors/        # Kafka, S3, Cron connectors
        event_router.go    # Routes events to functions
```

Infrastructure Starter Code (OpenTelemetry Setup):

```go
// internal/extensions/observability/tracing.go

package observability

import (
    "context"
    "go.opentelemetry.io/otel"
    "go.opentelemetry.io/otel/attribute"
    "go.opentelemetry.io/otel/exporters/jaeger"
    "go.opentelemetry.io/otel/propagation"
    "go.opentelemetry.io/otel/sdk/resource"
    sdktrace "go.opentelemetry.io/otel/sdk/trace"
    semconv "go.opentelemetry.io/otel/semconv/v1.17.0"
    "go.opentelemetry.io/otel/trace"
)

// InitTracer initializes OpenTelemetry tracing with Jaeger exporter.
// Call this once at application startup.
func InitTracer(serviceName, jaegerEndpoint string) (func(context.Context) error, error) {
    exp, err := jaeger.New(jaeger.WithCollectorEndpoint(jaeger.WithEndpoint(jaegerEndpoint)))
    if err != nil {
        return nil, err
    }

    tp := sdktrace.NewTracerProvider(
        sdktrace.WithBatcher(exp),
        sdktrace.WithResource(resource.NewWithAttributes(
            semconv.SchemaURL,
            semconv.ServiceName(serviceName),
            attribute.String("environment", "production"),
        )),
    )
    otel.SetTracerProvider(tp)
    otel.SetTextMapPropagator(propagation.NewCompositeTextMapPropagator(
        propagation.TraceContext{},
        propagation.Baggage{},
    ))

    return tp.Shutdown, nil
}

// TraceContextFromRequest extracts trace context from HTTP headers.
func TraceContextFromRequest(r *http.Request) map[string]string {
    ctx := otel.GetTextMapPropagator().Extract(r.Context(), propagation.HeaderCarrier(r.Header))
```

```
    carrier := propagation.HeaderCarrier{}
    otel.GetTextMapPropagator().Inject(ctx, carrier)
    return map[string]string(carrier)
}

// AddTraceContextToInvocationRequest adds trace context to an InvocationRequest.
func AddTraceContextToInvocationRequest(req *types.InvocationRequest, traceCtx map[string]string) {
    if req.TraceContext == nil {
        req.TraceContext = make(map[string]string)
    }
    for k, v := range traceCtx {
        req.TraceContext[k] = v
    }
}
```

Core Logic Skeleton Code (Workflow Executor):

```go
// internal/extensions/workflows/executor.go

package workflows


import (

    "context"

    "time"

)


// WorkflowExecutor manages the execution of a workflow definition.

type WorkflowExecutor struct {

    workflowDef *WorkflowDefinition

    stateStore  StateStore

    router      gateway.Router // Use the existing router for function calls

    metrics     *types.Metrics

}


// Execute starts workflow execution and returns immediately (async execution).

// Use GetStatus to poll for completion.

func (e *WorkflowExecutor) Execute(ctx context.Context, input []byte) (string, error) {

    // TODO 1: Generate a unique execution ID for this workflow run

    // TODO 2: Persist initial state to StateStore with status "RUNNING"

    // TODO 3: Start a goroutine to actually run the workflow steps

    // TODO 4: Return the execution ID to the caller

    return "", nil

}


// runWorkflow is the internal goroutine that executes workflow steps.

func (e *WorkflowExecutor) runWorkflow(ctx context.Context, execID string, input []byte) {

    // TODO 1: Load workflow definition and current state

    // TODO 2: Start from the step defined in workflowDef.StartAt

    // TODO 3: For each step:

    //   - If type is "function", invoke the function via router.Route

    //   - If type is "parallel", invoke all branches concurrently

    //   - If type is "choice", evaluate conditions to select next step

    // TODO 4: Handle step errors using the Catch/Retry policies

    // TODO 5: Update state store after each step completion

    // TODO 6: When workflow completes (or fails), update final status

}


// GetStatus returns the current status of a workflow execution.

func (e *WorkflowExecutor) GetStatus(ctx context.Context, execID string) (*WorkflowStatus, error) {

    // TODO 1: Retrieve execution state from StateStore

    // TODO 2: Return current step, overall status, and any outputs collected so far
```

```
    return nil, nil
}

// StateStore interface abstracts persistence for workflow state.
type StateStore interface {
    SaveState(ctx context.Context, execID string, state *WorkflowState) error
    LoadState(ctx context.Context, execID string) (*WorkflowState, error)
}
```

Language-Specific Hints (Go):

- Use `context.Context` to propagate trace spans and cancellation across goroutines.
- For custom domains, consider using the `gorilla/mux` router for advanced path matching.
- For GPU support, use `github.com/NVIDIA/nvidia-container-runtime` to interface with Docker.
- For event sources, use `github.com/segmentio/kafka-go` for Kafka connectivity and `github.com/robfig/cron/v3` for cron scheduling.

Milestone Checkpoint (Testing Custom Domains): After implementing custom domain routing:

1. Start the gateway with the enhanced router.
2. Configure a domain mapping: `api.test.local -> test-function`.
3. Use curl to test: `curl -H "Host: api.test.local" http://localhost:8080/any/path`
4. Verify the request is routed to `test-function` and the `InvocationRequest.Source` field contains the original host and path.
5. Check logs to confirm the correct routing decision.

Debugging Tips for Extensions:

Symptom	Likely Cause	How to Diagnose	Fix
Custom domain requests return 404	Routing table not populated	Check that domain mappings are loaded into router at startup	Ensure configuration is loaded before gateway starts
Workflow steps execute out of order	Race conditions in parallel steps	Add detailed logging of step start/end times	Use `sync.WaitGroup` to wait for all parallel steps before proceeding
Trace spans are not connected	Context propagation missing in internal calls	Check if `TraceContext` is being passed in `InvocationRequest`	Ensure all internal HTTP calls include trace headers
GPU function fails to start	NVIDIA runtime not installed on worker node	Check container logs for "could not select device driver"	Install NVIDIA container toolkit on worker nodes
Multi-cloud instance unhealthy	Network connectivity between control plane and worker	Check if control plane can reach worker endpoint	Configure proper security groups and VPN tunnels

Glossary

> **Milestone(s):** This reference section defines key terminology used throughout the design document, providing clarity for concepts spanning all milestones (1-5).

The serverless function runtime design introduces many specialized terms and concepts. This glossary provides clear definitions to ensure consistent understanding across the team and document.

General Serverless Concepts

Term	Definition	Context & Examples
Cold Start	The latency incurred when a function invocation requires the creation and initialization of a new execution environment from scratch. This includes loading the runtime, dependencies, and function code before execution can begin.	Occurs when no warm instances are available and a new container/microVM must be provisioned. Measured as the time from invocation request to when the handler begins executing.
Warm Start	A function invocation that uses a pre-initialized execution environment from the warm pool, avoiding the overhead of environment provisioning and initialization.	Happens when a previously used instance is reused. Typically 10-100x faster than cold starts.
Isolation	The architectural principle of preventing one function from observing, interfering with, or exhausting resources allocated to another function, ensuring security and predictable performance in multi-tenant environments.	Implemented through container namespaces, cgroups, seccomp, and/or microVMs. Critical for preventing "noisy neighbor" problems.
Handler	The entry point function written by developers that processes invocation requests. The runtime loads and invokes this function when a request arrives.	In Go: `func HandleRequest(ctx context.Context, payload []byte) ([]byte, error)`. Specified in the `FunctionDefinition.Handler` field.
Artifact	A self-contained package containing function code and all its dependencies, ready for deployment and execution in the runtime environment.	Created during packaging (Milestone 1). For Go, this is a statically linked binary; for Python, a zip with code and virtualenv.
Immutable Versioning	A versioning scheme where once a function version is created, it cannot be modified. Each change produces a new version with a unique identifier, enabling rollbacks and consistent deployments.	Implemented using content-addressable storage where the artifact hash becomes the version identifier.
Control Plane	The system components responsible for orchestration, scaling, and management decisions—coordinating what should happen but not directly processing requests.	Includes the Controller, Scaler, and WarmPoolManager. Makes decisions about when to create/destroy instances.
Data Plane	The system components responsible for actual request processing and function execution—handling the "data path" of invocations.	Includes the Gateway, Router, and Worker instances. Processes user requests end-to-end.
Multi-tenancy	The ability of the system to securely run functions from multiple customers (tenants) on shared infrastructure while maintaining isolation between them.	Achieved through strong isolation boundaries at container/microVM level and per-tenant resource quotas.
Ephemeral Storage	Instance-local persistent storage that lasts for the lifetime of a function instance, typically mounted at `/tmp`, which is cleaned up when the instance terminates.	Used for temporary files, caching, or any data that doesn't need to persist across invocations. Different from durable object storage.
Event Sources	External systems that trigger function invocations, such as message queues, storage bucket events, HTTP gateways, or scheduled timers.	The runtime's HTTP Gateway is one event source; future extensions could add S3-compatible storage events or Kafka integration.

Isolation & Security Technologies

Term	Definition	Context & Examples
MicroVM	A lightweight virtual machine optimized for serverless workloads, providing hardware-enforced isolation with minimal overhead. Firecracker is the canonical example.	Offers stronger security than containers with similar startup times when using snapshot restore. Used when maximum isolation is required.
cgroups (Control Groups)	A Linux kernel feature that limits, accounts for, and isolates the resource usage (CPU, memory, disk I/O, network) of a collection of processes.	Used to enforce memory limits (`memory.limit_in_bytes`) and CPU quotas (`cpu.cfs_quota_us`). Critical for preventing resource exhaustion attacks.
Namespaces	A Linux kernel feature that partitions kernel resources such that one set of processes sees one set of resources while another set of processes sees a different set, providing isolation for processes.	Includes PID (process ID), network, mount, UTS (hostname), IPC, and user namespaces. Containers use these to isolate processes from the host.
seccomp (Secure Computing)	A Linux kernel feature that restricts the system calls a process can make, effectively creating a sandbox at the syscall level.	Used with a restrictive profile that only allows necessary syscalls (read, write, exit) and blocks dangerous ones (ptrace, reboot).
AppArmor	A Linux Security Module (LSM) that allows system administrators to restrict programs' capabilities with per-program profiles, providing mandatory access control.	Can be used to restrict file access, network capabilities, and library loading beyond what namespaces provide.
Security Hardening	The process of securing a system by reducing its attack surface and eliminating vulnerabilities through configuration, least-privilege principles, and defensive coding.	Includes using minimal base images, dropping capabilities, setting no-new-privileges, and applying seccomp/AppArmor profiles.
Zombie Containers	Containers that weren't properly cleaned up and continue consuming system resources (memory, PIDs) despite having exited or being orphaned.	Caused by improper lifecycle management. The `ContainerManager` must ensure `RemoveContainer` is called even if `StopContainer` fails.
OOM Killer (Out-Of-Memory Killer)	A Linux kernel component that terminates processes when the system is critically low on memory, selected based on heuristics and oom_score.	Triggered when a function exceeds its memory limit without proper cgroup constraints. Proper cgroup configuration ensures the function process itself is killed, not other system processes.

Packaging & Dependencies

Term	Definition	Context & Examples
Dependency Resolution	The process of determining all required libraries and their compatible versions for a function, handling conflicts and ensuring reproducible builds.	For Python: reading `requirements.txt` and resolving transitive dependencies. For Go: running `go mod tidy` to update `go.mod` and `go.sum` .
Content-Addressable Storage (CAS)	A storage system where content is retrieved by its cryptographic hash rather than location or name, providing deduplication and integrity verification.	Used for storing function artifacts. The SHA-256 hash of the artifact becomes its address, enabling immutable versions and efficient caching.
Dependency Hell	A situation where different functions or versions require incompatible versions of the same library, causing conflicts that prevent proper execution.	Mitigated by isolated execution environments (each function gets its own dependencies) and careful version pinning during packaging.
Language-Specific Runtime Bundling	The process of packaging function code with its language runtime and dependencies into a self-contained executable or archive.	Go: compiled to static binary. Python: virtualenv zipped with handler. Java: Uber JAR with all dependencies.
Base Image	A minimal container image containing just the language runtime and essential system libraries, used as the foundation for function execution environments.	Examples: `golang:alpine` for Go, `python:3.9-slim` for Python. The function artifact is injected into this base image at runtime.
Overlay Filesystem	A union filesystem that layers a writable layer on top of read-only base layers, allowing multiple containers to share the same base image while having individual writeable spaces.	Used to efficiently share common runtime layers across function instances while giving each instance its own `/tmp` directory.

Execution & Optimization

Term	Definition	Context & Examples
Warm Pool	A collection of pre-initialized function instances ready for immediate use, maintained to reduce cold start latency for incoming requests.	Managed by the `WarmPoolManager`. Contains instances in `InstanceStateWarm` state, waiting for assignment.
Instance Reuse	The practice of returning a container to the warm pool after execution completes (if healthy) for reuse by subsequent invocations, avoiding teardown/creation overhead.	Controlled by the `ReleaseInstance` method with a `healthy` boolean indicating whether the instance can be reused.
Predictive Warming	Proactively creating warm instances based on traffic pattern analysis, historical data, or scheduled events, anticipating demand before it arrives.	The `PreWarm` method creates instances ahead of expected load. More advanced than reactive scaling.
Snapshot/Restore	The technique of capturing a running container's state (memory, process tree, open files) to disk and later restoring it to running state much faster than starting from scratch.	Implemented with CRIU (Checkpoint/Restore In Userspace). Can achieve sub-100ms restores for prepared environments.
Just-In-Time (JIT) Compilation Caching	Caching the results of runtime compilation (e.g., Python bytecode, JVM JIT optimizations) across invocations to reduce execution latency.	For Python: `.pyc` files cached in the container's filesystem. For Java: HotSpot JVM profile data persisted across warm starts.
Instance Cleanup	The process of resetting an execution environment to a fresh state before reuse, removing any temporary files, resetting network connections, and clearing memory.	Implemented by the `RuntimeCleaner` interface. Ensures no data leaks between invocations on the same instance.
Provisioned Concurrency	A configuration that keeps a minimum number of instances always warm, guaranteeing zero cold starts for those instances up to the provisioned level.	Useful for latency-sensitive workloads. Implemented as `MinWarmInstances` in the `PoolConfig`.

Routing & Load Management

Term	Definition	Context & Examples
Least-Connections	A load balancing strategy that sends incoming requests to the instance with the fewest active connections/requests, promoting even distribution of load.	Used by the `Router.selectInstance` method. Helps balance load across instances with varying execution times.
Circuit Breaker	A design pattern that prevents requests from being sent to unhealthy instances by temporarily "opening the circuit" after a threshold of failures is reached.	Implemented in `functionRoutingTable.circuitBreaker`. Stops routing to an instance that consistently fails, giving it time to recover.
Request Queue	A bounded buffer that holds incoming requests when all available instances are busy, preventing immediate rejection during traffic spikes.	Implemented as a priority queue in `RequestQueue`. Configurable maximum size to prevent memory exhaustion.
Timeout Propagation	The practice of passing deadline information from the client through all system layers to the function execution, ensuring consistent timeout handling across the stack.	The `InvocationRequest.Deadline` is set by the gateway and respected by the router, executor, and container manager.
Concurrency Limit	The maximum number of simultaneous requests a function instance can handle, preventing over-subscription and resource contention.	Enforced per instance (`routedInstance.activeReqs`) and per function (`functionRoutingTable.concurrencyLimit`).
Sticky Routing	Sending subsequent requests from the same client/session to the same function instance, useful for maintaining in-memory state or warming caches.	Optional feature using `InvocationRequest.PreferredInstanceID`. Implemented with a TTL to handle instance termination.
Load Shedding	Intentionally rejecting requests (with 503 Service Unavailable) when the system is overloaded, protecting critical components from collapse under excessive load.	Implemented when the request queue is full or when system-wide resource thresholds are exceeded.
Backpressure	A mechanism that signals upstream components to slow down when downstream components are overloaded, preventing cascading failures.	The request queue depth metrics can feed back to the gateway to adjust admission control.

Scaling & Lifecycle

Term	Definition	Context & Examples
Auto-Scaling	The dynamic adjustment of function instances based on demand, automatically adding instances during load increases and removing them during decreases.	Implemented by the `Scaler` component, which evaluates metrics and makes `ScaleDecision`s.
Scale-to-Zero	The capability to remove all running instances of a function after a period of inactivity (idle timeout), eliminating resource consumption when not in use.	Triggered by `Scaler` when `IdleTimeout` is reached. All instances transition through `InstanceStateDraining` to `InstanceStateTerminated`.
Target Concurrency	The desired number of concurrent executions per instance used as a scaling target; scaling occurs to maintain actual concurrency near this target.	Set in `ScalingConfig.TargetConcurrency`. If actual concurrency per instance exceeds this, scale up; if below, scale down.
Scaling Thrashing	Rapid oscillation between scale-up and scale-down actions caused by overly sensitive thresholds or insufficient stabilization periods.	Prevented by cooldown periods (`ScaleUpCooldown`, `ScaleDownCooldown`) and hysteresis (different up/down thresholds).
Cooldown Period	A minimum time that must elapse between scaling actions of the same direction, preventing rapid successive adjustments and allowing metrics to stabilize.	Configurable per function in `ScalingConfig`. The `Scaler` tracks `lastScaleUpTime` and `lastScaleDownTime`.
Idle Timeout	The duration an instance must be idle (no active requests) before being considered for termination during scale-down operations.	Different from TTL (absolute maximum lifetime). An instance may be terminated earlier due to scale-down even if not idle.
Hysteresis	Using different thresholds for scale-up versus scale-down to create a "dead band" that prevents oscillation around a single threshold.	Example: scale up at 70% utilization, scale down at 30% utilization, preventing thrashing at 50% utilization.
Reactive Scaling	Scaling based on current metrics (e.g., current concurrency, queue depth), responding to observed load changes.	Simpler to implement. The default approach in the `Scaler.EvaluateScaling` method.
Predictive Scaling	Scaling based on predicted future demand using historical patterns, time series forecasting, or scheduled events.	More complex but can pre-warm instances before load arrives. Future extension using machine learning models.

State Management & Events

Term	Definition	Context & Examples
Lifecycle Event	A discrete occurrence in the system that triggers state transitions, such as instance creation, request assignment, or timeout expiration.	Represented by `InstanceEvent` with `Type`, `Timestamp`, and related data. Published via `EventDispatcher`.
State Machine	A component that manages state transitions with validation, ensuring an entity moves through predefined states according to business rules.	The `StateMachine` for `FunctionInstance` validates transitions using `validateTransition` and executes handlers for each state.
Flow Coordinator	An orchestrator component that coordinates cross-component interactions during complex workflows like invocation handling with fallbacks.	The `FlowCoordinator` manages the end-to-end flow of a request, handling timeouts, retries, and error recovery.
Event Dispatcher	A component that distributes events to subscribers using channels or callbacks, enabling loose coupling between components.	The `EventDispatcher` allows components like `Scaler`, `WarmPoolManager`, and metrics collectors to react to events.
Transition Validation	The process of checking whether a state change is allowed according to defined rules before executing the transition.	The `StateMachine.validateTransition` method ensures illegal transitions (e.g., `Terminated` → `Active`) are rejected.
Drain Timeout	The maximum time allowed for graceful shutdown of an instance, during which it completes in-flight requests but accepts no new ones.	When scaling down, instances transition to `InstanceStateDraining` and have until the drain timeout to finish.
Graceful Degradation	The system's ability to provide reduced functionality when under stress or partial failure, rather than completely failing.	Example: When storage is unavailable, the system might accept invocations but disable function uploads, rather than rejecting all requests.

Error Handling & Resilience

Term	Definition	Context & Examples
Defense-in-Depth	A layered approach to error handling and security with multiple fallbacks, so if one mechanism fails, others still provide protection.	Example: Functions run in containers (layer 1) with cgroup limits (layer 2) and seccomp filters (layer 3).
Error Propagation	The pattern of passing error information through system components with appropriate context, enabling meaningful logging and recovery decisions.	`ClassifiedError` wraps raw errors with type, retryability, and context, flowing from executor to gateway.
Exponential Backoff	A retry strategy where the delay between retries increases exponentially, reducing load on recovering systems and preventing retry storms.	Used when retrying failed invocations or container operations. Configurable with `BackoffBase` and `BackoffMax`.
Fail-Stop	A design where components stop completely on failure (with clear error signals) rather than continuing in an undefined or corrupted state.	Preferable in isolation components: if a container cannot be securely created, fail completely rather than run with weakened isolation.
Cold Start Retry	Specifically retrying an invocation that failed due to cold start timeout, giving a newly created instance more time to initialize.	Controlled by `SyncInvokerConfig.RetryOnColdStart`. The `handleColdStartRetry` method determines if retry is appropriate.
Recovery State Machine	A specialized state machine that manages the recovery lifecycle of a failed instance, attempting cleanup, health checks, and potential restart.	Future extension: when an instance enters `InstanceStateError`, a recovery state machine could attempt automated repair.

Testing & Observability

Term	Definition	Context & Examples
Arrange-Act-Assert	A unit test pattern with clear separation of test setup (Arrange), execution (Act), and verification (Assert), improving test readability.	Used throughout the test suite. Example: Arrange a mock container manager, Act by calling `CreateContainer`, Assert that container exists.
Property-Based Tests	Tests that generate random inputs according to specified properties to discover edge cases, rather than testing specific predefined examples.	Useful for testing the router's load balancing algorithm with random request distributions and instance health states.
Golden Files	Reference test output files stored alongside tests for comparison, ensuring output format stability across refactorings.	Used for testing serialization formats (e.g., `FunctionDefinition` JSON marshaling) and CLI command outputs.
Testcontainers	A pattern for spinning up real container dependencies (like databases) in tests, providing more realistic integration testing.	Used in integration tests that require actual Docker containers rather than mocks.
Deterministic Time	A controllable time source (like `FakeClock`) used in timing-sensitive tests, allowing precise control over time-based behavior.	Critical for testing timeout logic, TTL expiration, and scaling cooldowns without actually sleeping.
Distributed Tracing	End-to-end tracking of requests across service boundaries using unique trace IDs and span IDs, enabling performance analysis in complex systems.	Implemented via OpenTelemetry with Jaeger. `TraceContextFromRequest` extracts and propagates trace context.
Trace Context	A set of headers (trace ID, span ID, sampling flags) that propagate across services for distributed tracing, following the W3C Trace Context standard.	Added to `InvocationRequest` via `AddTraceContextToInvocationRequest` and passed through all components.
Observability	The measure of how well internal states of a system can be inferred from its external outputs (logs, metrics, traces), beyond simple monitoring.	The runtime provides metrics (Prometheus), structured logs, and distributed traces for full observability.

Advanced Concepts & Extensions

Term	Definition	Context & Examples
Custom Domains	Mapping user domain names (e.g., `api.example.com`) to specific functions with advanced routing rules, TLS termination, and custom middleware.	Future extension: adds a domain manager component that integrates with the gateway for host-based routing.
Function Composition	Orchestrating multiple functions into workflows where the output of one function becomes the input to another, creating serverless applications.	Implemented via the `WorkflowEngine` interpreting `WorkflowDefinition` with steps, error handlers, and retry policies.
Workflow Engine	A component that interprets workflow definitions and manages stateful execution of steps, handling retries, error branching, and parallel execution.	Manages `WorkflowState` persistence and coordinates function invocations as defined in `WorkflowStep` configurations.
GPU Support	Allocating and exposing graphics processing units (or other accelerators) to functions for machine learning, video processing, or scientific computing.	Requires specialized container configuration (nvidia-docker), GPU-aware scheduling, and billing for accelerator time.
Specialized Hardware	Accelerators like GPUs, FPGAs, or TPUs allocated to functions for compute-intensive workloads beyond general-purpose CPUs.	Future extension requiring hardware inventory management, driver isolation, and specialized scheduling algorithms.
Multi-Cloud Deployment	Running the runtime across multiple cloud providers and/or on-premises data centers, with unified management and function portability.	Architecture designed with pluggable components (storage, container runtime) to support different infrastructure backends.
Edge Deployment	Running function instances geographically close to end-users for reduced latency, with synchronization challenges for state and code distribution.	Requires lightweight worker nodes, efficient artifact distribution, and location-aware routing in the gateway.

> **Key Insight:** Consistent terminology is crucial in distributed systems. This glossary not only defines terms but reveals the conceptual framework of the runtime—how isolation, speed, and efficiency trade-offs are managed through specific technologies and patterns. When implementing, ensure your team uses these exact terms to avoid confusion.

Common Acronyms

Acronym	Full Form	Context
CAS	Content-Addressable Storage	Used for artifact storage (Milestone 1)
CRIU	Checkpoint/Restore In Userspace	Used for container snapshot/restore (Milestone 3)
OOM	Out-Of-Memory	Error type when function exceeds memory limits
RPS	Requests Per Second	Key scaling metric (Milestone 5)
TTL	Time-To-Live	Maximum lifetime for warm instances
JIT	Just-In-Time	Compilation caching optimization
JVM	Java Virtual Machine	Runtime for Java functions
VM	Virtual Machine	General virtualization technology
LSM	Linux Security Module	AppArmor, SELinux security frameworks
API	Application Programming Interface	How users interact with the system
HTTP	Hypertext Transfer Protocol	Primary invocation protocol
JSON	JavaScript Object Notation	Data serialization format
CLI	Command Line Interface	Administrative tooling
UI	User Interface	Web-based management console (future)
IPC	Inter-Process Communication	Namespace type for container isolation
UTS	UNIX Timesharing System	Namespace for hostname isolation
PID	Process ID	Namespace for process tree isolation