# Architectural Analysis Report: Chris Abbott Implementation vs. Naive default implementation

**Executive Summary**

This report presents a comprehensive comparison between two implementations of a backend proxy service: the new server-based architecture and the vanilla naïve implementation you would expect from a 2-4 hour go at the problem.

Both approaches serve as intermediaries to country-specific company information backends, but they differ significantly in architecture, robustness, and scalability.

The Robust Implementation represents a significant architectural upgrade, adopting enterprise patterns for resilience, concurrency control, and endpoint management. While the Naive implementation provides basic functionality with a simpler design, the Robust Implementation offers superior performance under load, better error handling, and more sophisticated endpoint lifecycle management.

## 1. Architectural Overview

### 1.1 Naive Implementation

The Naive implementation follows a simple, monolithic design:

- **Core Structure**: A single Service struct managing all functionality
- **Request Handling**: Direct HTTP handlers without concurrency control
- **Backend Mapping**: Simple mapping of country codes to backend URLs
- **Caching**: Basic TTL-based cache with global scope

[HTTP Request] → [Service Handler] → [Backend Request] → [Response Transformation] → [HTTP Response]

### 1.2 Robust Implementation

The Robust Implementation adopts a multi-layered architecture with clear separation of concerns:

- **Core Components**:
  - Server: HTTP interface and request routing
  - WorkerPool: Concurrency control and job management
  - Endpoint: Backend interface with state management
- **Request Flow**: HTTP request → job queue → worker processing → endpoint call → response
- **Multi**-flow: Fast-pass for cache hits

- **Concurrency Model**: Controlled parallelism with worker pools

```
[HTTP Request] → [Server] → [Job Queue] → [Worker Pool] → [Endpoint]
→ [Backend] → [Response]
```

**2. Concurrency and Scalability**

**2.1 Naive Implementation**

- Creates a new goroutine for each incoming request (implied by HTTP handlers)
- No limits on concurrent backend requests
- Could lead to resource exhaustion under heavy load
- No request buffering or prioritization

**2.2 Robust Implementation**

- Uses a worker pool pattern with a configurable number of workers
- Job queue to buffer incoming requests
- Controlled concurrency with MaxWorkers parameter
- Explicit queue timeout handling
- Better resource utilization under load
- More resilient to traffic spikes

**Scalability Comparison**

| **Aspect** | **Naive** | **Robust Implementation** |
|---|---|---|
| Concurrent Requests | Unlimited (potential resource exhaustion) | Controlled by worker pool |
| Request Buffering | None | Job queue with configurable capacity |
| Resource Control | Minimal | Explicit with worker limits |
| Traffic Spike Handling | Poor | Good with queuing mechanisms |

**3. Endpoint Management**

**3.1 Naive Implementation**

- Backend endpoints represented as URL strings
- Direct mapping of country codes to backend URLs
- No endpoint health tracking or lifecycle management
- Fixed timeout (900ms) for all backend requests

### 3.2 Robust Implementation

- Formal Endpoint objects with state machines
- Sophisticated endpoint health tracking:
    - StatusActive: Normal operation
    - StatusInactive: Temporary failure with retry logic
    - StatusDead: Permanent failure
- Exponential backoff retry strategy
- Proactive health checking during initialization
- Per-endpoint configuration (SLA, cache size)

**Endpoint State Management Comparison**

| Feature | Naive | Robust Implementation |
|---|---|---|
| Health Tracking | None | 3-state (Active, Inactive, Dead) |
| Retry Mechanism | None | Exponential backoff |
| Timeout Control | Fixed (900ms) | Configurable per endpoint |
| Recovery | Manual intervention | Automatic with retry logic |

## 4. Caching Strategy

### 4.1 Naive Implementation

- Custom cache implementation with a map and mutex
- Global cache scope for all backends
- Fixed 24-hour TTL for all cache entries
- Thread-safe via read/write mutex

### 4.2 Robust Implementation

- Uses LRU (Least Recently Used) cache library
- Per-endpoint cache instances
- Configurable cache size via application configuration
- Proper cache lifecycle management during shutdown

**Caching Comparison**

| Feature | Naive | Robust Implementation |
|---|---|---|
| Cache Type | Fixed TTL | LRU (Least Recently Used) |

| Feature | Naive | Robust Implementation |
|---|---|---|
| Cache Scope | Global | Per-endpoint |
| Eviction Strategy | Time-based only | Access-based (LRU) |
| Configuration | Fixed 24h TTL | Configurable size |

## 5. Error Handling and Resilience

### 5.1 Naive Implementation

- Basic error handling with HTTP status codes
- Errors passed directly to clients
- No automatic recovery from endpoint failures
- No distinction between temporary and permanent failures

### 5.2 Robust Implementation

- Multi-layered error handling
- Context-based cancellation throughout request lifecycle
- Detailed error logging at each stage
- Endpoint-specific error processing with state transitions
- Ability to recover from temporary failures

**Error Handling Comparison**

| Feature | Naive | Robust Implementation |
|---|---|---|
| Error Granularity | Basic HTTP status | Detailed error types |
| Failure Recovery | None | Automatic with retries |
| Timeout Handling | Simple request timeout | Multi-level with context |
| Error Reporting | Minimal | Comprehensive logging |

## 6. Content Type Handling

### 6.1 Naive Implementation

- Handles application/x-company-v1 and application/x-company-v2 content types
- Type-specific response structs (BackendV1Response, BackendV2Response)
- Conversion to standard CompanyResponse model

### 6.2 Robust Implementation

- Similar content type handling approach
- Formal content type constants

- Type-specific response structs with conversion methods
- Consistent model conversion

**Content Type Handling Comparison**

| **Feature** | **Naive** | **Robust Implementation** |
|---|---|---|
| Content Types | V1 and V2 | V1 and V2 with formal constants |
| Response Parsing | Direct unmarshaling | Structured with validation |
| Model Conversion | In-handler transformation | Separate conversion methods |

**7. Resource Management**

**7.1 Naive Implementation**

- Limited explicit resource management
- No formal shutdown procedure
- Potential for resource leaks under error conditions

**7.2 Robust Implementation**

- Explicit resource cleanup with Close method for endpoints
- Cache purging during shutdown
- Clear component lifecycle management
- Controlled startup and shutdown sequences

**8. Performance Implications**

**8.1 Naive Implementation**

- Simpler with less overhead for low-traffic scenarios
- No queuing latency
- Potential for unchecked resource consumption under load
- More vulnerable to cascading failures

**8.2 Robust Implementation**

- Better resource utilization under load
- More resilient to traffic spikes
- Controlled backend pressure
- Better timeout handling to prevent resource leaks
- Slight queuing latency for high-traffic scenarios

**9. Maintainability and Extensibility**

### 9.1 Naive Implementation

- Simpler, more straightforward code
- Less separation of concerns
- Harder to extend with new features
- Testing is more challenging due to tight coupling

### 9.2 Robust Implementation

- Clear separation of concerns with component interfaces
- More modular design facilitates extension
- Better testability with clear component boundaries
- More complex but more maintainable for large codebases

## 10. Conclusion and Recommendations

The Robust Implementation represents a significant architectural improvement over the Naive implementation, particularly for production environments with varying traffic patterns and backend reliability.

### Key Advantages of Robust Implementation

1. **Resource Control**: Better management of system resources through worker pools and queuing
2. **Reliability**: Comprehensive error handling and endpoint health management
3. **Scalability**: Controlled concurrency and better handling of traffic spikes
4. **Maintainability**: Clean separation of concerns and abstraction layers
5. **Observability**: More detailed logging and health monitoring

### Use Case Recommendations

- **Naive Implementation**: Suitable for development environments, simple deployments, or scenarios with consistent low traffic and highly reliable backends.
- **Robust Implementation**: Recommended for production environments, scenarios with variable traffic patterns, or when dealing with potentially unreliable backend services.

The Robust Implementation demonstrates several enterprise-level patterns that make it significantly more robust and production-ready, making it the preferred choice for mission-critical deployments where service reliability is paramount.

## Original design notes:

Before I started coding, I broke down the problem in my brain to formulate a spec.

Notes and thoughts

<<<

> It should get the list of backends from the command line at startup. The format is a list of "iso-code=backend-address" mappings, for example: ./your-solution ru=http://localhost:9001 us=http://localhost:9002. Backendify has only one backend per country as of now, so the ISO country codes are guaranteed to be unique.

>>>

Sounds like a very long command line!

Have got to find out what URLs are being sent in.

Health test should check whether those URLs are actually alive on startup, maybe check country codes.

Healthcheck could also check status of worker pool whenever called and return a too many requests error

**Initial checks upon application start**

Do the endpoints respond?

Are the country codes valid?

If any country codes invalid – fatal error – because they're probably meant to be something else and that would be missing in the production data

Endpoints not functional on app start: start app but endpoints marked. Exponential retry should be on different schedule. Health endpoint response time should not depend on checking back end availability except on application start (503) because that would time it out and make the server appear to be offline with disastrous consequences.

**Cacheing**

Country code/company_id

Can this be in process memory? What's the concurrency needed here?

Cache should be available to each golang process

LRU Cache appropriate – memory capped

Purge process – linked lists with date-times so prune is fast? Let's see how big the cache gets in the research before engineering that.

**Timeouts of 1 second**

Adhering to the SLA vs returning meaningful information

Possible sources of timeouts

  not enough resources available for request – minimised via cacheing and ensuring SLA adherence at all times.

  timeout from provider – we know we're waiting, but need to return a response. This is a "couldn't do request" rather than a "couldn't find company", but our API specification doesn't cover that. Letting the request naturally time out feels horrible and results in even more aggressive traffic. I think you'd have to have our API return a HTTP 500 error as well to cover this, possibly put onto a global 0.99 second timeout so that processes were never left hanging (even if the API client drops the connection, the process is still running, waiting for things). Put a flag in so that we can configure the error returned on SLA timeout or endpoint unavailability– 404 or 500. Have configurable retry on endpoints

**Things we don't know**

What's the cache hit/miss rate for companies/countries?

What are the kinds of errors returned by each endpoint and does it matter? (probably not)

How often does the load balancer call the status endpoint?

In the absence of any other data we need to deploy a data collection version of the application first and log appropriately.

**Testing**

Unit test

Component tests – happy path, unhappy path (company not found), unhappy path (backend takes too long), unhappy path (backend is marked as down)

Mocking – v1 endpoint mock with configurable delay, v2 endpoint mock with delay. Use Mockery? Easiest to set these up as separate services on different ports within the build rather than set up different Docker images with different projects

Configurable list of mock servers to be created based on input country list – this is basically an "internal Mountebank" approach, but with greatly simplified.

Component test start code would set up mock services, but would require

**Endpoint – main purpose, hold endpoint details, and status**

**** Application state, should be available to all processes – might be bottleneck ****

Country code:

Endpoint URL:

Status: <AVAILABLE>, <UNAVAILABLE>

Status changed timestamp:

(you might put reliability metrics here, but that's not for now)

**Concurrency**

What's the expected request rate?

How many parallel backend requests should each instance handle?

Should we use Go's goroutines, or a worker-pool pattern? Deciding on worker-pool because of SLA requirement and variable length of requests. Also, pool can be monitored as part of health check to warn off load balancer.

**Solution API design**

The API customers are expecting is quite simple.

Your application must implement just two endpoints:

1. GET /status. This API endpoint must return an HTTP 200/OK when your solution is ready to accept requests. The load balancer will use that endpoint to monitor your solution in production.

Status could check the endpoint objects: but is that a criteria for a healthcheck? This is basically just a READY. The only time this healthcheck would fail are (a) cache down, (b) all endpoints down,  or (c) application still initializing and checking. Would this status ever change? It might if there was a theoretical limit on the number of processes running that had been hit, but you would assume the load balancer was checking these health aspects outside the application. What error do you return when it's not ready?

2. GET /company?id=XX&country_iso=YY. This API endpoint receives the parameters id and country_iso. id can be a string without any particular limitation. country_iso will be a two-letter country code to select the backend according to the application configuration. Your solution must query the backend in a proper country and return:
    - An HTTP 200/OK reply when the company with the requested id exists on the corresponding backend. The body should be a JSON object with the company data returned by a backend.
    - An HTTP 404/Not Found reply if the company with the requested id does not exist on the corresponding backend.

Your solution should always reply with the following JSON object to the customer:

```
{
  "id": "string, the company id requested by a customer",
  "name": "string, the company name, as returned by a backend",
  "active": "boolean, indicating if the company is still active according to the active_until date",
  "active_until": "RFC 3339 UTC date-time expressed as a string, optional."
}
```

**Backend providers API description**

As of now, there are only two backend variants, V1 and V2. Backendify, in compliance with the industry's best practices, is in a state of transition between the two backends, so your solution must support both.

Both backends will answer HTTP GET requests on the /companies/<id> path, where id is the arbitrary string. However, their replies are slightly different:

1. V1 backend will return the JSON object of the following format:

```
{
 "cn": "string, the company name.",
 "created_on": "RFC3339 UTC datetime expressed as a string.",
 "closed_on": "RFC3339 UTC datetime expressed as a string, optional.",
}
```

2. V1 backend reply will have a Content-Type of an application/x-company-v1.
3. V2 backend will return the JSON object of the following format:

```
{
 "company_name": "string, the company name.",
 "tin": "string, tax identification number",
 "dissolved_on": "RFC3339 UTC datetime expressed as a string, optional.",
}
```

4. V2 backend reply will have a different Content-Type of an application/x-company-v2.

My assumption: there may be rubbish data in the replies but we will ignore that data on the JSON unmarshal.

**Running in Production**

This repository has a continuous delivery setup. Every time you push code to the repo, it will deploy your solution to the production environment as a single instance. The orchestrator will give it at least 1 CPU, 128MB of RAM, and a volatile 1G drive located at /tmp.

When your solution reports that it is ready to serve requests (with the /status endpoint described above), the load balancer will unleash customer traffic to it.

Things to consider when dealing with the backends:

1. They are located in distant areas. They are not reliable. They can time out, return errors, or throttle you. Some of them are in dire need of an upgrade and are not particularly fast.

Do we get granular enough looking at data from backends to have different strategies for time-outs (could be environmental at our end, but probably not), 4xx errors (including authentication/permission errors unlikely to be fixed soon), 5xx errors (might be temporary, or connected to the individual query, or due to throttling.

2. Your solution has an SLA when replying to a customer: 95% of requests should reply within 1 second. If your application does not answer within an SLA, the customers will consider this an error, abort the request, and retry aggresively a number of times.
3. To account for unreliable backends and still stay within an SLA, your solution might use caching. You can cache any correct reply from the backend for 24 hours as data is really slow to change.

This doesn't help hugely if the cache is empty 😊

**Accessing production to debug**

You can directly access the logs in the canary deployment.

In production, there is no direct access to logs due to the extremely high load. At thousands of requests per second, the engineering team decided to provide a StatD-compatible metrics server instead, and your solution can use up to five metrics. Check the FAQ for more information.

Security checks: out of scope

DDoS protection: out of scope