The Senior Go Engineer Interview Guide: AI Platform Engineering

Production-Grade Go, LLM Platforms, RAG, Vector Search, and Cloud Native Systems

v1.0.0

Luca Sepe

Index

Senior Go Engineer Interview Guide: AI Platform Engineering

How to Use This Book

This book is a practical preparation guide for senior Go engineering interviews.

It focuses on the reasoning expected from an experienced engineer: trade-offs, failure modes, operational behavior, and clear communication.

The chapters move from Go fundamentals to concurrency, distributed systems, Kubernetes, observability, and system design. Each question contains four parts:

1. **Question**: a realistic interview prompt.
2. **What the interviewer is testing**: the signal behind the question.
3. **Answer**: a concise response suitable for an interview.
4. **Senior follow-up**: deeper considerations that distinguish a production engineer from someone who only knows the definitions.

A Strong Interview Answer

A strong senior-level answer usually follows this order:

1. State the principle or invariant.
2. Describe the simplest correct design.
3. Identify trade-offs and failure modes.
4. Explain how you would observe and test it.
5. Ask for constraints before optimizing.

For example, when asked how to make an API resilient, do not begin with a list of products. Begin with the failure model:

> I would first define which dependencies can fail, how long callers can wait, whether operations are safe to retry, and what degraded behavior is acceptable.

That answer shows engineering judgment. Naming a circuit breaker without explaining what it protects does not.

Suggested Study Plan

First pass: foundations

Read Chapters 02 through 06. Run every Go example and modify it. In particular, practice explaining:

- who owns each goroutine;
- who closes each channel;
- how cancellation propagates;
- which data is shared;
- how errors retain useful context.

Second pass: systems reasoning

Read Chapters 07 through 10. For every design, draw:

- request flow;
- state ownership;
- retry boundaries;
- consistency guarantees;
- partial failure behavior.

Third pass: production operations

Read Chapters 11 through 16. Practice moving between application-level and platform-level explanations. A senior Go developer should be able to explain why a request timed out in the handler, at the load balancer, or during Kubernetes scheduling.

Fourth pass: AI platform engineering

Read Chapters 17 through 21. Trace one request from tenant admission through retrieval, model streaming, observability, and cost accounting. Practice explaining which controls are correctness boundaries and which are quality optimizations.

Fifth pass: capstone

Complete Chapter 22 at the terminal. Run every request, inspect every package, and perform at least two failure exercises. Practice presenting the project without claiming that its deterministic teaching components are production AI models.

Final pass: mock interviews

Use Chapter 23 without notes. Answer each prompt aloud in two to five minutes. Then review the answer rubric.

Using the Companion Code

The executable examples live in the companion code repository:

```
github.com/lucasepe/the-senior-go-engineer-interview-guide-ai-platform-engineering
```

The `companion code` directory contains executable examples for Chapters 02 through 22. From the repository root, verify all examples with:

```
go test ./...
go test -race ./...
go vet ./...
```

Do not study the examples by memorizing syntax. For each one, explain:

- the invariant it protects;
- which component owns mutable state;
- how concurrent work terminates;
- which failure behavior is intentionally omitted;
- what must change before using the design in production.

Several examples use an in-memory store to keep the relevant invariant visible.

Their chapter README states how that invariant maps to a database, broker, or Kubernetes environment.

Conventions Used in the Code

The examples target modern, idiomatic Go:

- `context.Context` is passed explicitly as the first parameter.
- Errors are wrapped with `%w` when callers may need the cause.
- Channels communicate ownership or coordinate work; they are not used merely because the language provides them.
- Goroutines always have a documented termination path.
- Readability is preferred over compactness.
- Comments explain contracts and non-obvious decisions.

Unless a question explicitly asks for a custom implementation, prefer the standard library. Production code should be unsurprising.

What Senior Means Here

Seniority is not measured by obscure syntax. It is the ability to:

- make constraints explicit;
- choose a design proportionate to the problem;
- predict operational failure;
- protect correctness under concurrency;
- build useful observability;
- communicate trade-offs without pretending certainty.

The best answer is rarely "always use X."

It is usually "given these constraints, I would choose X, and here is what would make me reconsider."

Channels and Context

1. What is the difference between buffered and unbuffered channels?

Answer: A send on an unbuffered channel completes only when a receiver is ready, creating a synchronization point. A buffered channel allows sends to complete while capacity remains.

A buffer can absorb short bursts or bound queued work. It does not make a slow consumer fast, and an arbitrarily large buffer can hide backpressure until memory is exhausted.

Senior follow-up: Buffer size should come from queueing expectations: acceptable wait time, arrival rate, processing rate, and memory cost per item.

Treat it as an operational parameter with metrics, not a magic constant.

2. Who should close a channel?

Answer: The sending side should close a channel when it can prove that no more values will be sent. Receivers generally should not close it because they do not own the sending lifecycle.

A channel does not need to be closed merely to release resources; garbage collection handles unreachable channels.

For multiple senders, coordinate them before closing:

```
func merge[T any](inputs ...<-chan T) <-chan T {
    output := make(chan T)
    var wg sync.WaitGroup
    wg.Add(len(inputs))

    for _, input := range inputs {
        input := input
        go func() {
            defer wg.Done()
            for value := range input {
                output <- value
            }
        }()
    }

    go func() {
        wg.Wait()
        close(output)
    }()

    return output
}
```

Senior follow-up: This merge function can leak if the consumer stops reading. A production version should accept a context and make each send cancellable.

3. How does `select` behave?

Answer: `select` waits until one case can proceed. If several are ready, one is chosen pseudo-randomly. A `default` case makes it non-blocking.

Sending to or receiving from a nil channel blocks forever, which can be used to disable a case.

Senior follow-up: Do not depend on `select` for strict fairness or priority. A loop with a `default` case can spin and consume CPU. Timer allocation inside hot loops can also create avoidable pressure.

4. What belongs in a context?

Answer: A context carries cancellation, deadlines, and request-scoped values across API boundaries.

It should not carry optional function parameters, long-lived dependencies, or mutable business state.

Guidelines:

- accept `context.Context` as the first parameter
- do not store it in a struct unless a framework contract requires it
- do not pass nil
- always call the cancel function returned by `WithCancel`, `WithTimeout`, or `WithDeadline`
- use private key types for context values

```
type requestIDKey struct{}

func WithRequestID(ctx context.Context, id string) context.Context {
    return context.WithValue(ctx, requestIDKey{}, id)
}
```

Senior follow-up: Context values are appropriate for cross-cutting metadata such as request IDs and trace context.

Required domain input should remain an explicit parameter so the API contract is visible.

5. How should deadlines propagate across services?

Answer: A downstream call should receive a deadline no later than the caller's deadline. The service should reserve enough time to process the response and return it, rather than spending the caller's entire budget downstream.

```
func loadProfile(ctx context.Context, client *http.Client, url string) error {
    downstreamCtx, cancel := context.WithTimeout(ctx, 500*time.Millisecond)
    defer cancel()

    req, err := http.NewRequestWithContext(downstreamCtx, http.MethodGet, url, nil)
    if err != nil {
        return fmt.Errorf("create profile request: %w", err)
    }

    resp, err := client.Do(req)
    if err != nil {
        return fmt.Errorf("get profile: %w", err)
    }
    defer resp.Body.Close()

    if resp.StatusCode != http.StatusOK {
        return fmt.Errorf("get profile: unexpected status %s", resp.Status)
    }
    return nil
}
```

Senior follow-up: Configure transport-level timeouts as well. Context deadlines bound a request, while dial, TLS handshake, response-header, and idle connection settings protect particular network phases and resource pools.

6. How would you implement a cancellable pipeline?

Answer: Every stage must observe cancellation both while receiving and while sending. Otherwise a downstream stage can stop and leave an upstream stage blocked forever.

```
func square(ctx context.Context, input <-chan int) <-chan int {
    output := make(chan int)

    go func() {
        defer close(output)
        for {
            select {
            case <-ctx.Done():
                return
            case value, ok := <-input:
                if !ok {
                    return
                }

                select {
```

```
                case output <- value * value:
                case <-ctx.Done():
                    return
                }
            }
        }
    }()

    return output
}
```

Senior follow-up: Pipelines can be elegant for streaming transformations, but ordinary synchronous functions are easier to reason about for small bounded collections.

Choose concurrency because it improves the system, not because it makes the code look concurrent.

7. What happens when you receive from a closed channel?

Answer: Receives return immediately. Buffered values are delivered first; after the buffer is drained, receives return the element type's zero value and `ok == false`.

```
value, ok := <-ch
if !ok {
    // The channel is closed and drained.
}
```

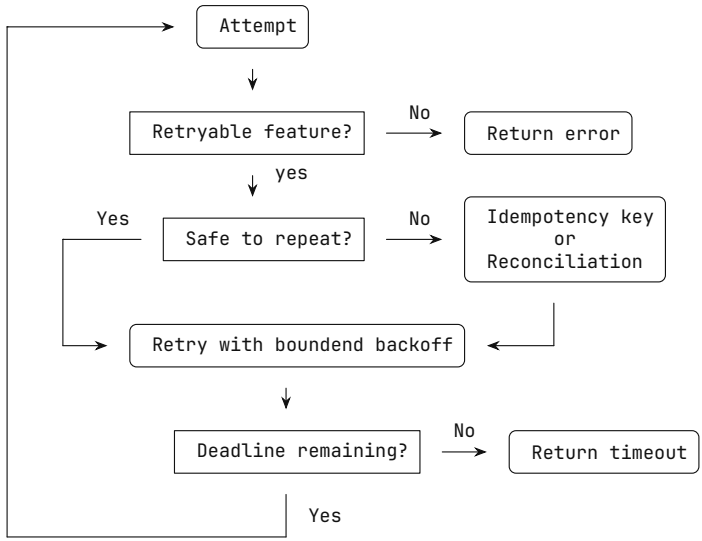
Sending to a closed channel panics. Closing an already closed channel also panics. Receiving from a nil channel blocks forever.

Senior follow-up: A zero value is not sufficient evidence that a channel is closed because it may be a valid message. Use the two-value receive or `range`.

Companion Code

See `04-channels-context` for a pipeline where every blocking channel operation observes cancellation.

Retries, Idempotency, and Timeouts



1. When should a service retry?

Answer: Retry only transient failures, only when the operation is safe to repeat, and only within a bounded time and attempt budget.

Examples may include connection resets, temporary unavailability, or selected rate-limit responses.

Do not retry validation errors, authentication failures, or permanent resource absence. Be careful with timeouts: the caller may not know whether the server committed the operation before the response was lost.

Senior follow-up: Retry at one well-chosen layer.

Nested retries multiply attempts and can turn a small incident into an overload event.

2. Why use exponential backoff with jitter?

Answer: Exponential backoff reduces request frequency during a sustained failure.

Jitter prevents many clients from retrying in synchronized waves.

```
func retry(
    ctx context.Context,
    attempts int,
    initialDelay time.Duration,
    operation func(context.Context) error,
) error {
    if attempts <= 0 {
        return errors.New("attempts must be positive")
    }

    delay := initialDelay
    var lastErr error

    for attempt := 1; attempt <= attempts; attempt++ {
        if err := operation(ctx); err == nil {
            return nil
        } else {
            lastErr = err
        }

        if attempt == attempts {
            break
        }

        jitter := time.Duration(rand.Int64N(int64(delay/2) + 1))
        timer := time.NewTimer(delay/2 + jitter)
        select {
        case <-timer.C:
        case <-ctx.Done():
            timer.Stop()
            return ctx.Err()
        }

        if delay < 5*time.Second {
            delay *= 2
        }
    }

    return fmt.Errorf("operation failed after %d attempts: %w", attempts, lastErr)
}
```

Senior follow-up: Production retry code also needs error classification, server-provided `Retry-After`, metrics, a maximum delay, and tests with an injectable clock or delay function.

3. What is idempotency?

Answer: An idempotent operation produces the same intended state when applied multiple times as when applied once.

HTTP `PUT` is intended to be idempotent; `POST` often is not unless the application adds an idempotency mechanism.

For payment creation, a client can send an idempotency key. The server atomically stores the key, request fingerprint, processing state, and final response. A duplicate with the same payload returns the stored result; reuse with a different payload is rejected.

Senior follow-up: The key store must have a defined scope and retention period.

Concurrency matters: two simultaneous requests with the same key must not both execute the side effect.

4. What is the difference between at-most-once, at-least-once, and exactly-once?

Answer:

- At-most-once may lose work but does not intentionally redeliver it
- At-least-once retries or redelivers, so work is not silently lost but duplicates are possible
- Exactly-once means the observable effect occurs once, which generally requires atomicity or deduplication at the effect boundary

Senior follow-up: A broker's "exactly once" feature has a scope. It may cover broker reads and writes but not an external database or email provider. Describe the end-to-end effect, not only the transport guarantee.

5. How do you choose timeouts?

Answer: Start from the caller's end-to-end latency budget. Allocate portions to queueing, each dependency, local processing, and response transmission. Base values on measured latency distributions and business requirements, then leave headroom.

Senior follow-up: A timeout without cancellation merely makes the caller stop waiting while work continues. Propagate cancellation and ensure clients and drivers honor it. Track both timeout counts and the work that continues after client cancellation.

6. What is retry amplification?

Answer: Retry amplification occurs when retries at several layers multiply the load. If three layers each make three attempts, one user request can produce up to 27 calls at the deepest dependency.

Senior follow-up: Establish a retry budget and ownership policy. Propagate a deadline, cap attempts, use circuit breaking or load shedding, and expose attempt counts in metrics and traces.

7. How would you make a queue consumer idempotent?

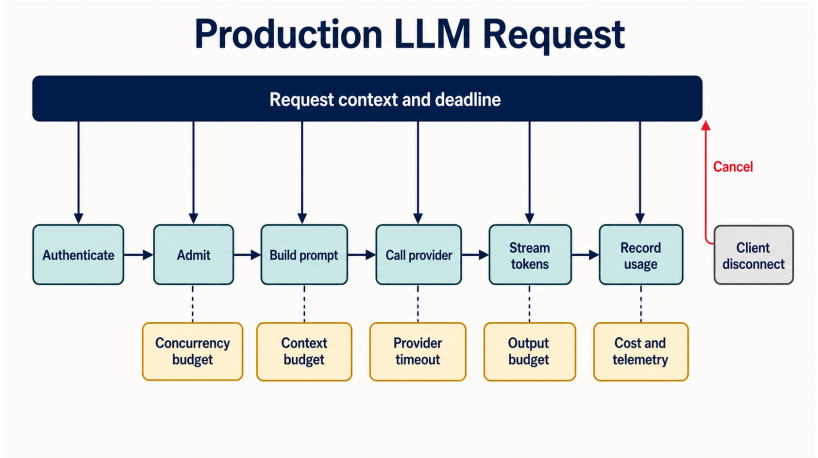
Answer: Identify a stable event ID and make deduplication atomic with the business effect. For example, insert the event ID into a table with a unique constraint in the same transaction that updates domain state. If the ID already exists, acknowledge the message without reapplying the effect.

Senior follow-up: Marking an event "processed" before the effect can lose work; marking it afterward can duplicate work. The transaction boundary solves this only when both records are in the same transactional store. External effects need their own idempotency key or an outbox.

Companion Code

See `08-retries-idempotency` for a classified retry policy and atomic handling of concurrent duplicate requests.

LLM Platform Engineering



1. What responsibilities belong in an LLM gateway?

Answer: A gateway can centralize provider authentication, model selection, request validation, deadlines, concurrency limits, token and cost budgets, streaming, retries, observability, and policy enforcement.

It should expose a stable internal contract without hiding provider capabilities callers genuinely need.

Senior follow-up: The gateway is a high-impact dependency.

Bound queues, isolate tenants and providers, support degradation, and avoid retrying ambiguous or non-idempotent operations blindly.

2. How do you stream model output safely in Go?

Answer: Tie the provider stream to the request context, make every channel or socket write cancellable, close the stream exactly once, and surface the final error separately from partial output.

Apply backpressure rather than buffering an unbounded response.

Senior follow-up: After response headers are sent, an HTTP status can no longer describe a later failure. Define an application-level terminal event or document truncated-stream behavior.

3. How should timeouts be allocated for inference?

Answer: Separate queue wait, provider connection, time to first token, and total generation time. Propagate the caller deadline and reserve time to encode and deliver the response.

Senior follow-up: Time to first token and inter-token delay reveal different failures. A single total-duration metric hides queue saturation and stalled streams.

4. How do you control LLM cost?

Answer: Enforce input and output token limits, per-tenant quotas, concurrency limits, model allowlists, caching where semantics permit, and explicit budgets.

Record usage and cost by stable operation and tenant class.

Senior follow-up: Reject before expensive work when possible.

Cost controls must also cover retries, embedding ingestion, reranking, background evaluation, and abusive high-cardinality requests, such as requests that create many unique keys, prompts, or cache entries.

5. When can an inference request be retried?

Answer: Retry connection failures or explicit transient provider responses before output is exposed, within the original deadline.

Once tokens have been streamed, transparent retry can duplicate or contradict output.

Senior follow-up: Provider request IDs and idempotency support may narrow ambiguity, but guarantees are provider-specific. Expose retry attempts and reason.

6. How do you handle provider fallback?

Answer: Define compatibility first: model capability, context length, tool calling, safety policy, latency, cost, and output contract. Fall back only when the product accepts changed behavior.

Senior follow-up: A fallback that silently changes quality is not resilience; it is an undocumented product decision. Version and evaluate routing policy.

7. What should be observed for an LLM operation?

Answer: Request rate, failures, queue time, time to first token, total latency, input and output tokens, truncation, provider and model, retry count, cancellation, and estimated cost.

Use sampling and redaction for prompts and responses.

Senior follow-up: Never put raw prompts, user IDs, or document text in metric labels. Trace metadata should be privacy-reviewed and access-controlled.

8. How do you version model behavior?

Answer: Treat model, system prompt, tools, retrieval configuration, and safety policy as one versioned behavior bundle.

Support controlled rollout, comparison, rollback, and reproducible evaluation.

Senior follow-up: A provider model name alone is insufficient provenance.

Record the complete configuration needed to explain an output without storing sensitive content unnecessarily.

Companion Code

See `17-llm-platform` for a cancellation-aware streaming gateway with token and concurrency budgets.