What's new in Go 1.24?

Scheduled for release in February 2025.

Meet your speaker

George (Yehor) Sereda

Golang & Blockchain Developer

- Worked at #1 Blockchain startup - Israel.
- Veteran SME 2023 & 2024 at Coursera.
- Head of Backend in the first blockchain- based social network - United Kingdom.
- Nowadays working at the Mediahuis.

Agenda

1. Performance Optimisation
2. Language features
3. Standard Library
4. Security
5. Go vet
6. Questions

Performance Optimisation

1 **CPU load**

Reduced by 2-3%.

2 **Memory Allocation**

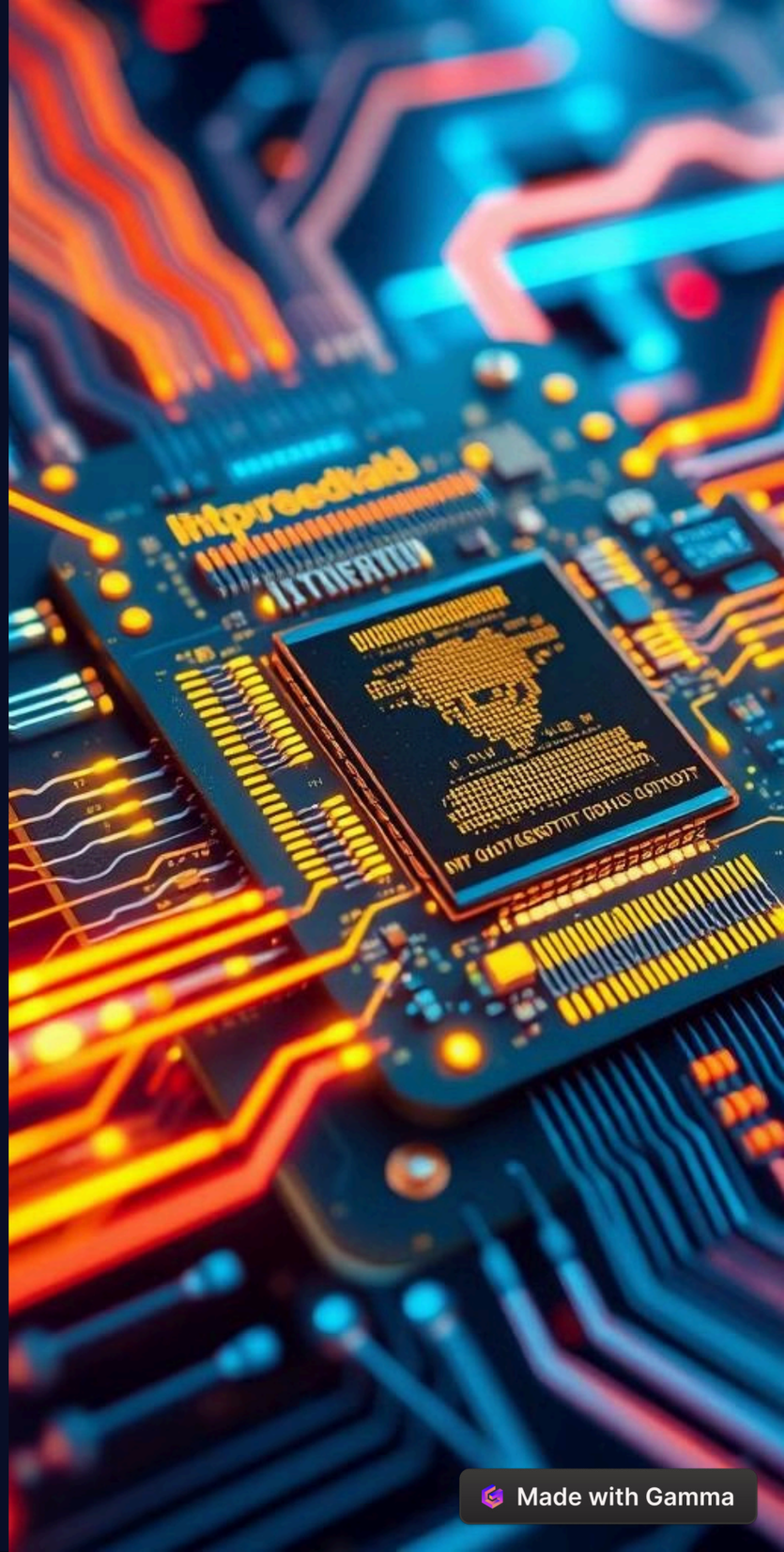
For small objects

3 **CGO**

Optimised interfacing with C libraries.

4 **Mutex**

5 **Swiss Tables in Map**

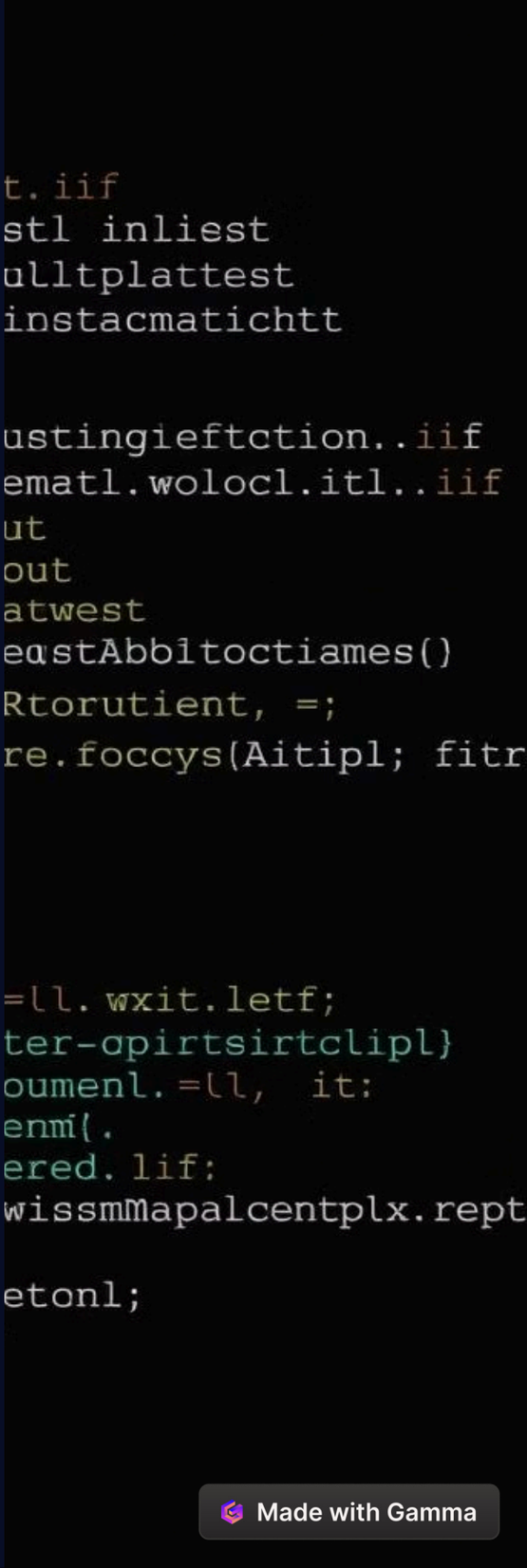


Swiss Tables: How It Works?

	Previous Implementation	Swiss Tables
Storage	Keys & Value's in buckets	Keys & Value's and metadata in arrays
Collisions	Uses linked lists	Uses linear probing
Lookup	Through traversing linked lists	Through Metadata to find the key quickly

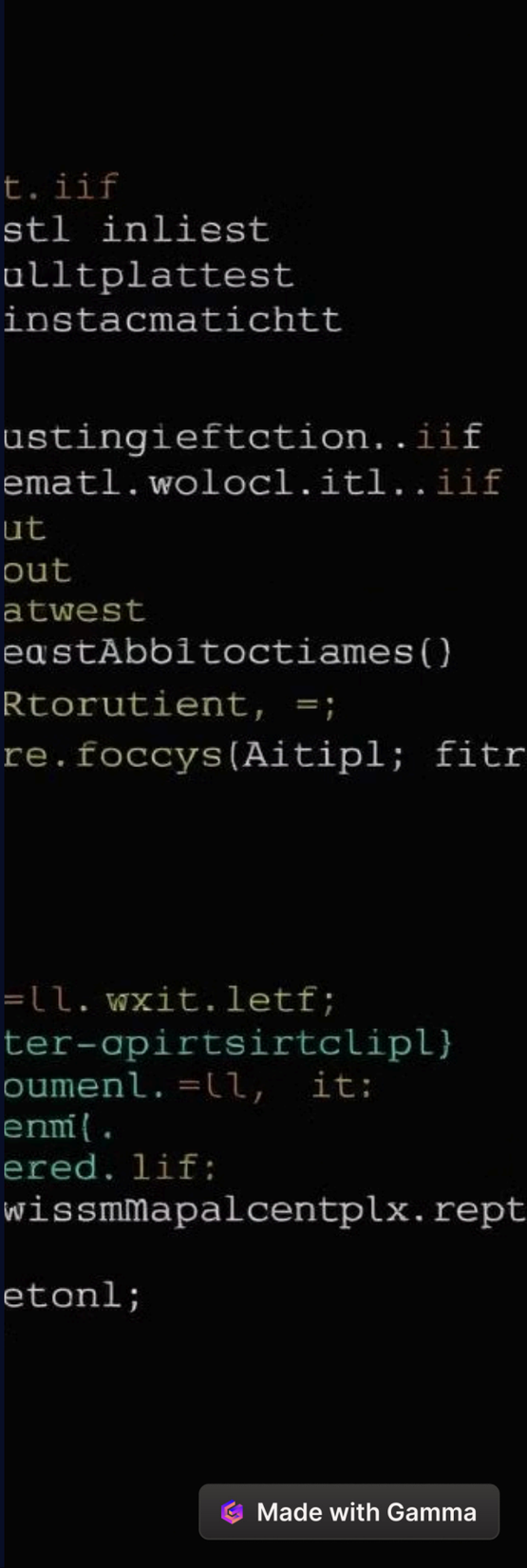
Swiss Tables: Code Example

```
type swissMap struct {
  keys []string
  values []int
  metadata []byte // example: [0, 0, 1, 0]
  mask int
}

func (m *swissMap) put(key string, value int) {
  hash := hashKey(key)
  index := hash & m.mask // initial bucket

  for m.metadata[index] != 0 { // Check for empty
    index = (index + 1) & m.mask // Probing on collision
  }

  m.keys[index] = key
  m.values[index] = value
  m.metadata[index] = byte(hash) // Store fingerprint
}
```



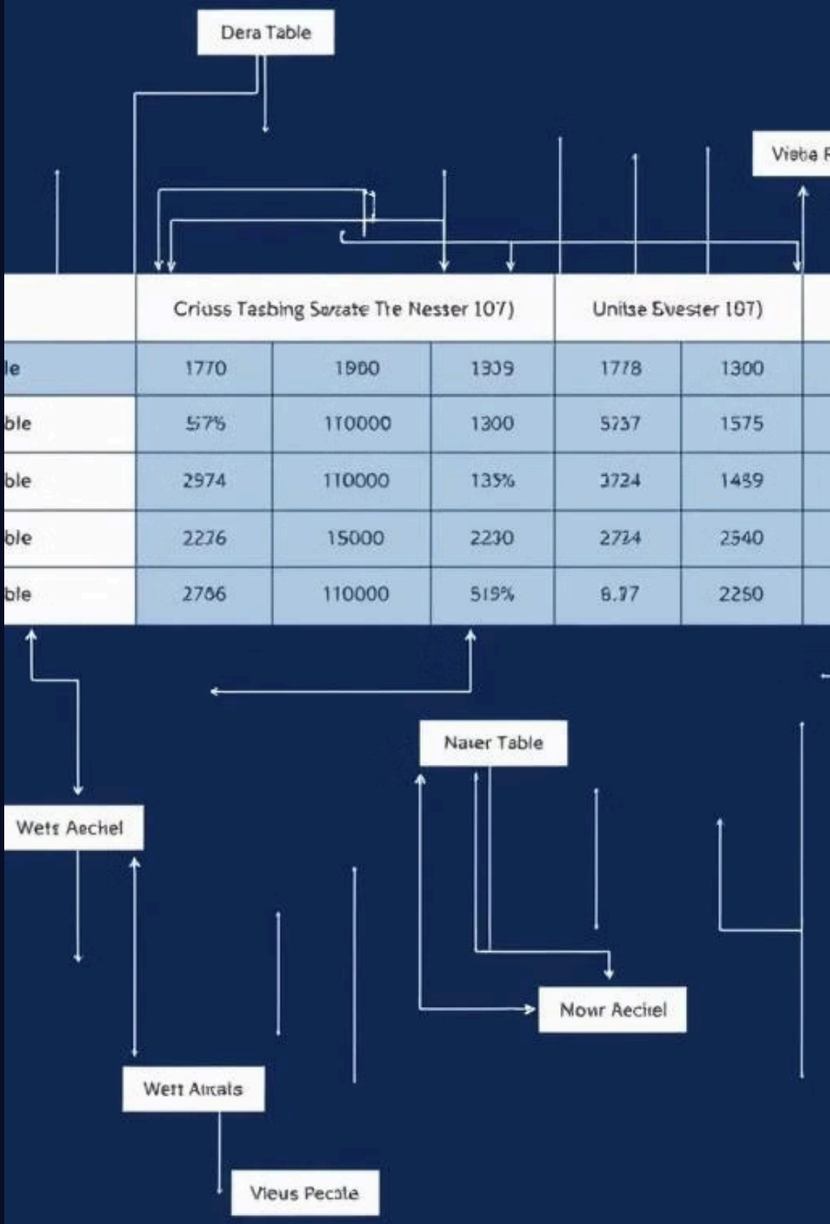
Swiss Tables: Code Example

```
type swissMap struct {
  keys []string
  values []int
  metadata []byte
  mask int
}

func (m *swissMap) get(key string) (int, bool) {
    hash := hashKey(key) & m.mask
    originalIndex := hash

    for m.keys[hash] != "" {
        if m.keys[hash] == key {
            return m.values[hash], true
        }
        hash = (hash + 1) & m.mask
        if hash == originalIndex {
            break
        }
    }
    return 0, false
}
```



Swiss Tables

1. Faster Lookups
2. Collision Handling
3. Reduced RAM Usage
4. Cache Performance



Language features

1. Weak Pointers
2. New Tool Directive
3. Generic Type Aliases

Weak Pointers

?

What?

Like pointer, but have no sence for the GC.

</>

Functions

```
func Make[T any](ptr *T) Pointer[T]
func (p Pointer[T]) Value() *T
```

New Tool Directive

Before

```
// tools.go
// +build tools

package tools

import (
  _ "github.com/rs/zerolog"
  _ "github.com/stretchr/testify"
)
```

After

```
// go.mod

module example.com/myproject

go 1.24

tool "github.com/rs/zerolog" v1.28.0
tool "github.com/stretchr/testify" v1.7.0
```

Generic Type Aliases

Before

```
package main

type Response[T any] struct {}

// Specific type aliases
type StringResponse = Response[string]
type UserResponse = Response[User]
```

After

```
package main

type Response[T any] struct {}

// Generic alias for Response
type GenericResponse[T any] = Response[T]
```

Standard Library

1. **T.Context**
   Tests context
2. **`omitzero` Tag**
   For JSON-marshalling
3. **os.Root**
4. **Iterator Functions**

os.Root

os.Root.Open // - opens a file for reading.
os.Root.Create // - creates a file.
os.Root.OpenFile // - is the generalized open call.
os.Root.Mkdir // - creates a directory.

Iterator Functions

```
func main() {
	str := "Hello, Go 1.24!"

	for _, r := range strings.NewReader(str) {
		fmt.Printf("%c ", r)
	}
}
```

Security

crypto/mlkem

Introduces Multi-Linear Key Encapsulation Mechanisms (MLKEM) for quantum computing resistance.

rand.Text()

Generates cryptographically secure random strings for tokens, IDs, and passwords.

Go vet

1. Misaligned Struct Fields
2. Slice Out-of-Bounds Access
3. Misused Context Cancellation

Go vet: Misaligned Struct Fields

Before

```
type Example struct {
    b bool // 1 byte
    i int // 8 bytes
    c byte // 1 byte, misaligned
}
```

Size of struct: 24 bytes

After

```
type Example struct {
    i int // 8 bytes
    b bool // 1 byte
    c byte // 1 byte, aligned
}
```

Size of struct: 16 bytes

Go vet: Slice Out-of-Bounds Access

```
s := []int{1, 2, 3}
fmt.Println(s[3]) // flagged as potential out-of-bounds access
```

Go vet: Misused Context Cancellation

Before

```
func main() {
    ctx, cancel := context.WithTimeout(
        context.Background(), time.Second)


    select {
      case <-time.After(2 * time.Second):

      case <-ctx.Done():
    }
}
```

After

```
func main() {
    ctx, cancel := context.WithTimeout(
        context.Background(), time.Second)
    defer cancel() // Proper cancellation

    select {
      case <-time.After(2 * time.Second):

      case <-ctx.Done():
    }
}
```

Key Takeaways

1

Performance Optimisation

Go 1.24 delivers significant performance improvements across various areas.

2

Language features

New language features enhance code readability and flexibility.

3

Standard Library

Updates to the standard library provide new functionalities and improvements.

4

Security

Enhanced security measures protect against emerging threats.

5

Go vet

Improved code analysis tools help identify potential issues and improve code quality.

Questions