LLM Agents in Production: Why Go is the Missing Piece

Kartik Ramesh

Design credits: Egon Elbre

Introduction

Some things about me:

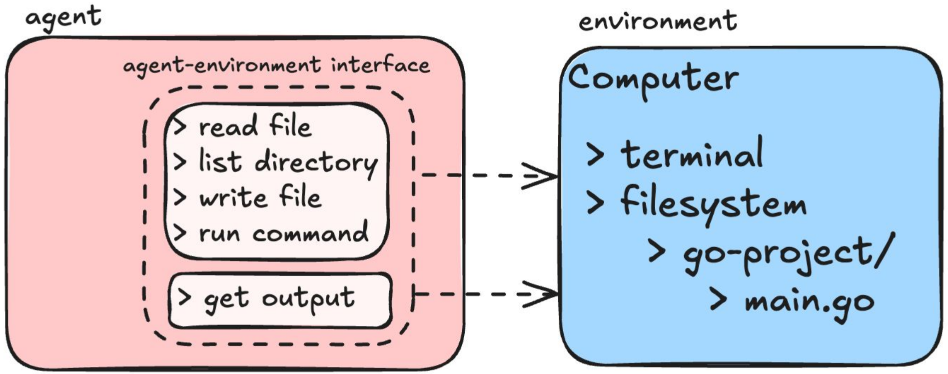
1. Grad student at UIUC
2. Researching LLM Inferencing Systems
3. Previously: Distributed Systems at Microsoft

First gophercon, hopefully of many

I’ve heard of agents!

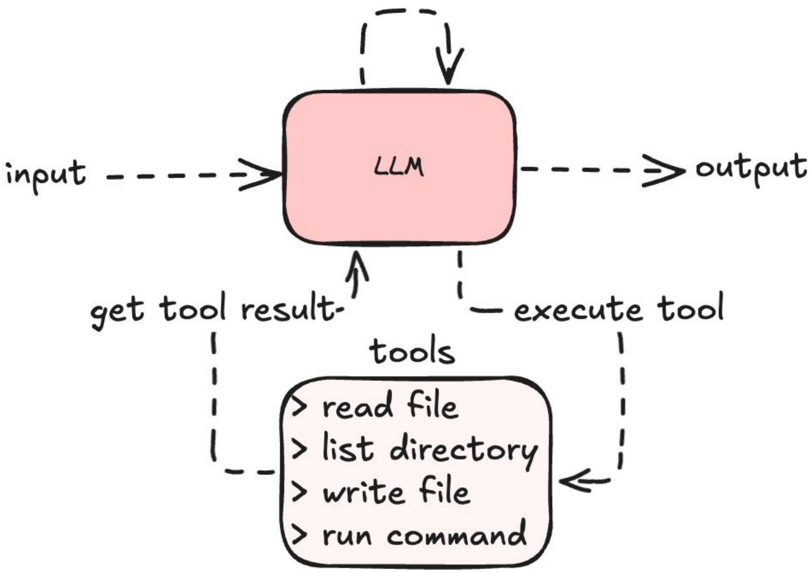
An agent is anything that can perceive its environment and act upon that environment.

An LLM agent is an LLM augmented with extra capabilities that allows it to act as an agent.

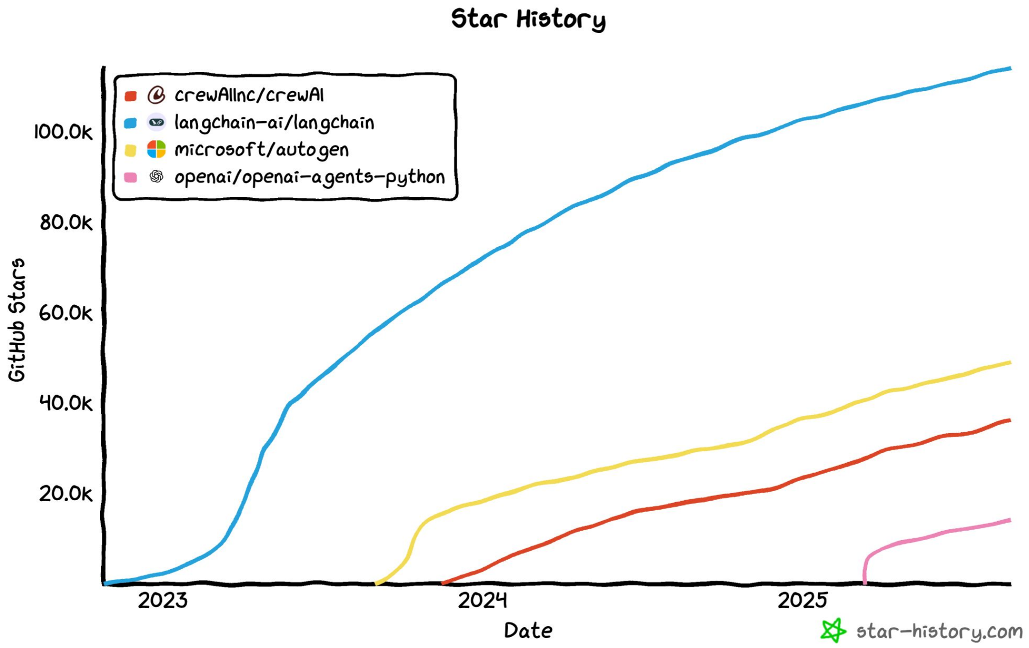


I’ve heard of agents!

LLM Agents = LLMs + tools in a while loop



Building agents



Why should you “Go” for agents?

Go offers unique advantages:

1. Concurrency: Goroutines handle multiple agent tasks with low overhead

2. Performance: Compiled binaries, efficient memory management

3. Production-Ready: Static binaries simplify containerization and Go's mature cloud-native ecosystem helps orchestrate agents at scale

Production deployment patterns are still emerging in this space - this is Go's opportunity!

Agenda

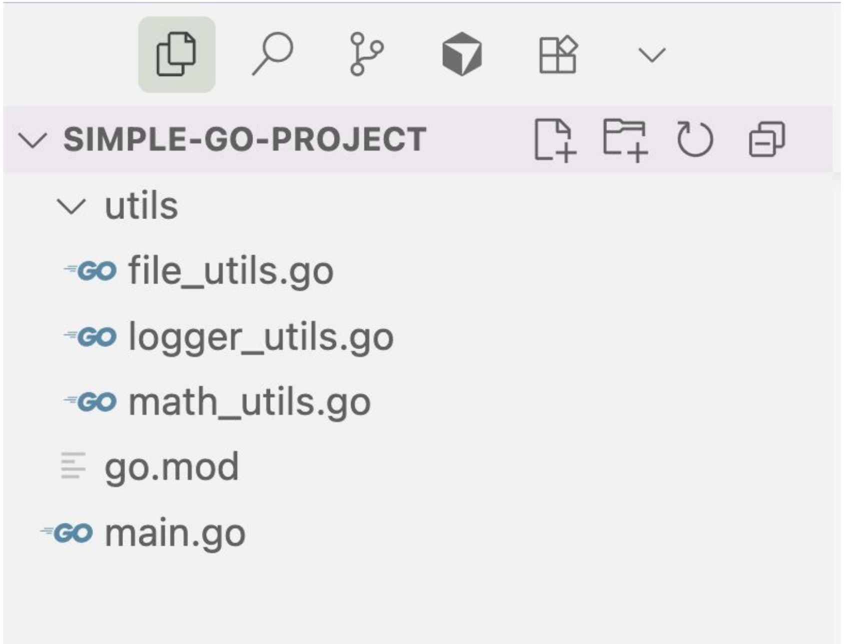
1. Build an LLM agent from scratch to write code for us.
2. Scale out to a multi-agent system
3. Containerizing and deploy our agents to production

Building a coding agent

Build an agent that can:

1. Explore a codebase
2. Update files to make code changes

We have a simple Go project with a TODO comment somewhere that the agent needs to implement.



Our simple LLM agent

```go
func main() {
    // uses ANTHROPIC_KEY env var
    client := anthropic.NewClient()
    reader := bufio.NewReader(os.Stdin)

    for {
        fmt.Printf("> ")
        input, _ := reader.ReadString('\n')

        response, _ := client.Messages.New(context.Background(),
            anthropic.MessageNewParams{
                Model: anthropic.ModelClaudeSonnet4_20250514,
                MaxTokens: 1024,
                Messages: []anthropic.MessageParam{
                    anthropic.NewUserMessage(
                        anthropic.NewTextBlock(input)),
                },
            })

        fmt.Printf("%s\n\n", response.Content[0].Text)
    }
}
```

Our simple LLM agent

```go
func main() {
    // uses ANTHROPIC_KEY env var
    client := anthropic.NewClient()
    reader := bufio.NewReader(os.Stdin)

    for {
        fmt.Printf("> ")
        input, _ := reader.ReadString('\n')

        response, _ := client.Messages.New(context.Background(),
            anthropic.MessageNewParams{
                Model: anthropic.ModelClaudeSonnet4_20250514,
                MaxTokens: 1024,
                Messages: []anthropic.MessageParam{
                    anthropic.NewUserMessage(
                        anthropic.NewTextBlock(input)),
                },
            })

        fmt.Printf("%s\n\n", response.Content[0].Text)
    }
}
```

Input to our agent

Our simple LLM agent

```go
func main() {
    // uses ANTHROPIC_KEY env var
    client := anthropic.NewClient()
    reader := bufio.NewReader(os.Stdin)

    for {
        fmt.Printf("> ")
        input, _ := reader.ReadString('\n')

        response, _ := client.Messages.New(context.Background(),
            anthropic.MessageNewParams{
                Model: anthropic.ModelClaudeSonnet4_20250514,
                MaxTokens: 1024,
                Messages: []anthropic.MessageParam{
                    anthropic.NewUserMessage(
                        anthropic.NewTextBlock(input)),
                },
            })

        fmt.Printf("%s\n\n", response.Content[0].Text)
    }
}
```

Our LLM call

Our simple LLM agent

```go
func main() {
    // uses ANTHROPIC_KEY env var
    client := anthropic.NewClient()
    reader := bufio.NewReader(os.Stdin)

    for {
        fmt.Printf("> ")
        input, _ := reader.ReadString('\n')

        response, _ := client.Messages.New(context.Background(),
            anthropic.MessageNewParams{
                Model: anthropic.ModelClaudeSonnet4_20250514,
                MaxTokens: 1024,
                Messages: []anthropic.MessageParam{
                    anthropic.NewUserMessage(
                        anthropic.NewTextBlock(input)),
                },
            })

        fmt.Printf("%s\n\n", response.Content[0].Text)
    }
}
```

Output

Demo time!

Okay, let’s see if 20 lines of code is what all the agentic hype is about.

Step 1. Define the Tool input

```go
// What our agent needs to pass in to use this tool.
type ReadFileInput struct {
    Path string `json:"path" jsonschema_description:"The path of the file."`
}
```

Step 2. Implement the tool function

```go
// ReadFile is a tool that reads the contents of a file.
func ReadFile(input json.RawMessage) (string, error) {
    var input ReadFileInput
    json.Unmarshal(input, &input)
    content, err := os.ReadFile(input.Path)
    return string(content), err
}
```

Step 3. Tell it to the LLM

```go
// Tell the LLM about our tool
var ReadFileTool = anthropic.ToolParam{
    Name: "read_file",
    Description: anthropic.String("Read the contents of a file"),
    InputSchema: GenerateSchema[ReadFileInput](),
}
```

Step 3. Tell it to the LLM

```go
response, _ := client.Messages.New(context.Background(),
    anthropic.MessageNewParams{
        Model: anthropic.ModelClaudeSonnet4_20250514,
        MaxTokens: 1024,
        Messages: messages,
        Tools: []anthropic.ToolUnionParam{{OfTool: &ReadFileTool}},
    })
```

Adding tools

```go
// Iterate over all messages in your response
for _, content := range response.Content {
    switch block := content.AsAny().(type) {
    case anthropic.TextBlock:
        fmt.Println(block.Text)
    case anthropic.ToolUseBlock:
        // If you have multiple tools use block.Name to find the function.
        result, _ := ReadFile(block.Input)
        // Aggregate multiple tool results.
        toolResults = append(toolResults, result)
    }

// Store results to pass to the LLM on the next iteration.
messages = append(messages, anthropic.NewUserMessage(toolResults...))
}
```

Adding tools

```go
// Iterate over all messages in your response
for _, content := range response.Content {
    switch block := content.AsAny().(type) {
    case anthropic.TextBlock:
        fmt.Println(block.Text)
    case anthropic.ToolUseBlock:
        // If you have multiple tools use block.Name to find the function.
        result, _ := ReadFile(block.Input)
        // Aggregate multiple tool results.
        toolResults = append(toolResults, result)
    }

// Store results to pass to the LLM on the next iteration.
messages = append(messages, anthropic.NewUserMessage(toolResults...))
}
```

Our better LLM agent

```go
type Agent struct {
    name string
    client *anthropic.Client
    tools     []ToolDefinition
    readInput func() (string, error)
    writeOutput func(string) error
    }
```

https://ampcode.com/how-to-build-an-agent

Demo 2

Okay that’s a lot of code, let’s see if this actually works.

Concurrent Tool Execution

```go
for _, content := range response.Content {
    switch block := content.AsAny().(type) {
    case anthropic.TextBlock:
        fmt.Println(block.Text)
    case anthropic.ToolUseBlock:
        result, _ := a.ExecuteTool(block.ToolID, block.ToolName, block.Input)
        toolResults = append(toolResults, result)
    }
}
```

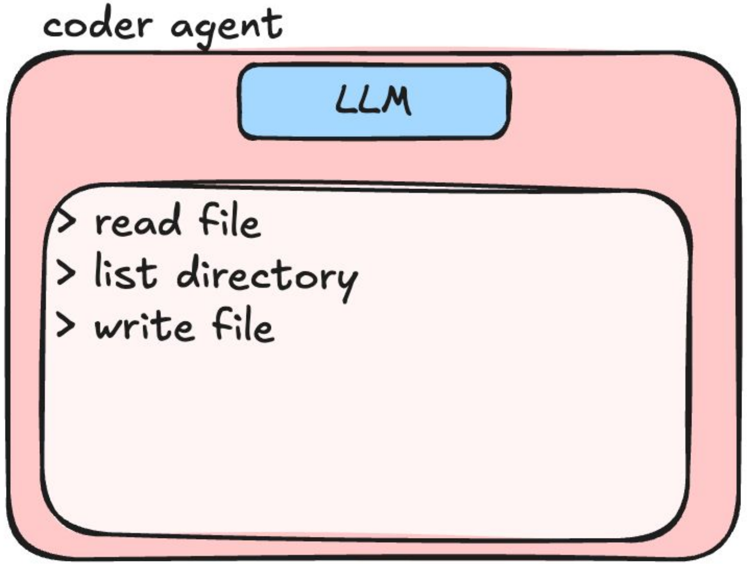
Concurrent Tool Execution with Goroutines

```diff
+ch := make(chan anthropic.ContentBlockParamUnion)
+toolCount := 0
+
    for _, content := range response.Content {
        switch block := content.AsAny().(type) {
        case anthropic.TextBlock:
            fmt.Println(block.Text)
        case anthropic.ToolUseBlock:
-           toolResult = a.ExecuteTool(block.ID, block.Name, block.Input)
-           toolResults = append(toolResults, toolResult)
+           toolCount++
+           go func() {
+               toolResult := a.ExecuteTool(block.ID, block.Name, block.Input)
+               ch <- toolResult
+           }()
        }
    }

+for i := 0; i < toolCount; i++ {
+    toolResults = append(toolResults, <-ch)
+}
```

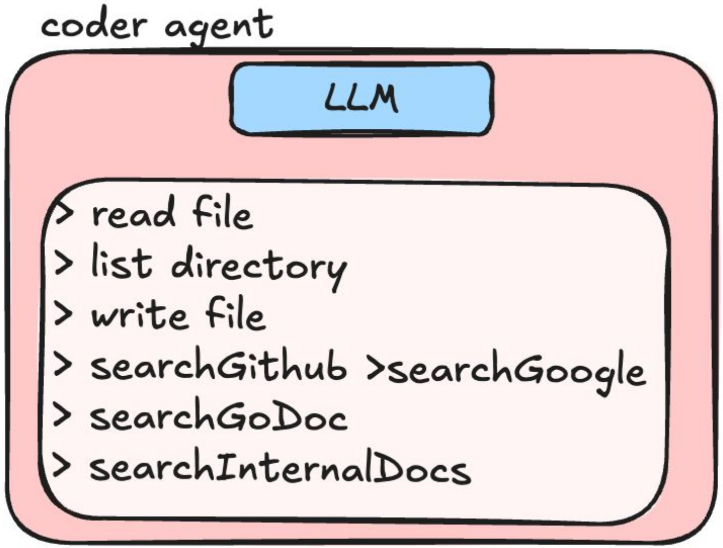
Concurrent Tool Execution with Goroutines

Demo: It can’t be that easy?

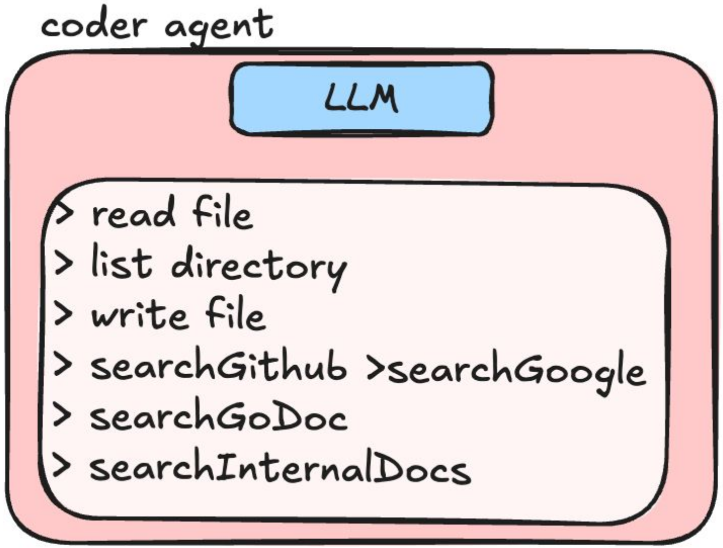
Extending capabilities



Extending capabilities



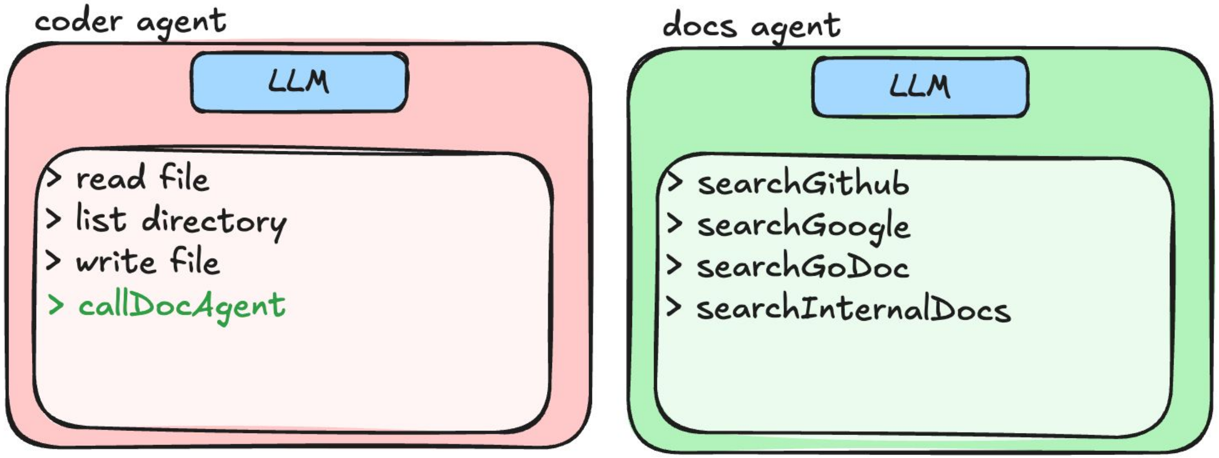
Extending capabilities



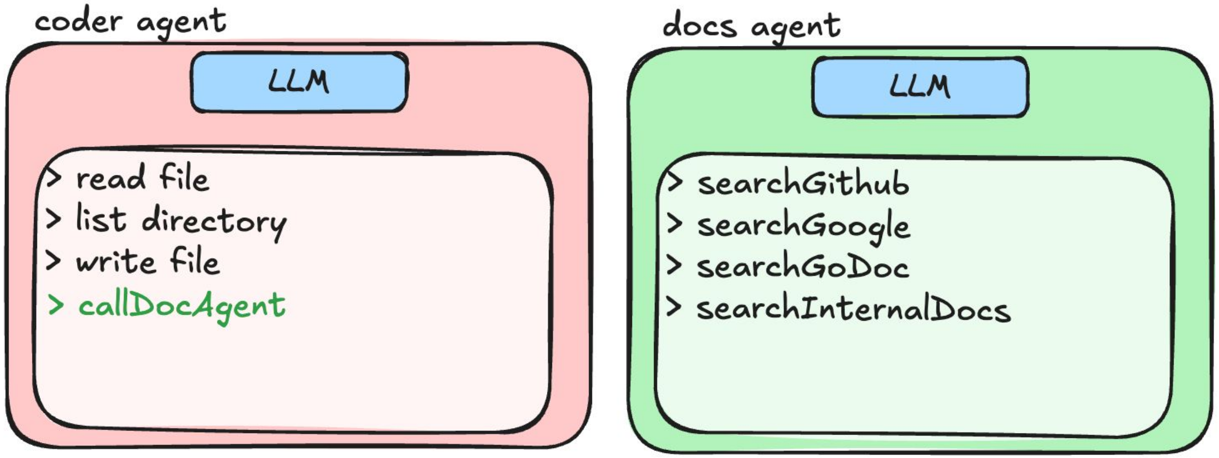
Problems with this approach:

1. Does not scale very well.
2. Every additional tool risks confusing our agent

Multi-agent systems



Multi-agent systems



Benefits:

1. Modularity
2. Specialization
3. Flexibility

Search Go Documentation

```go
// Search Go docs from pkg.go.dev
func SearchGoDocumentation(input json.RawMessage) (string, error) {
    var input SearchGoDocumentationInput
    json.Unmarshal(input, &input)

    url := fmt.Sprintf("https://pkg.go.dev/%s?tab=doc", input.PackageName)
    resp, _ := http.Get(url)

    doc, _ := goquery.NewDocumentFromReader(resp.Body)
    return doc.Find(".Documentation-overview").Text(), nil
}
```

Doc Agent

```go
// Specialized tools for documentation
var DocTools = []ToolDefinition{
    SearchGoDocumentationDefinition,
}

// DocAgent reads and writes to the network
docAgent := NewAgent(
    "docAgent",
    client,
    DocTools,
    ReadFromNetwork,
    WriteToNetwork,
)
```

Doc Agent

```go
// Specialized tools for documentation
var DocTools = []ToolDefinition{
    SearchGoDocumentationDefinition,
}

// DocAgent reads and writes to the network
docAgent := NewAgent(
    "docAgent",
    client,
    DocTools,
    ReadFromNetwork,
    WriteToNetwork,
)
```

Doc Agent

```go
// Specialized tools for documentation
var DocTools = []ToolDefinition{
    SearchGoDocumentationDefinition,
}

// DocAgent reads and writes to the network
docAgent := NewAgent(
    "docAgent",
    client,
    DocTools,
    ReadFromNetwork,
    WriteToNetwork,
)
```

```go
// Coder agent calls Doc agent over the network
func InvokeDocumentationAgent(input json.RawMessage) (string, error) {
    var input InvokeDocumentationAgentInput
    json.Unmarshal(input, &input)

    docAgentURL := os.Getenv("DOC_AGENT_URL") // Service discovery!
    payload := map[string]string{"query": input.Query}
    reqBody, _ := json.Marshal(payload)

    resp, _ := http.Post(docAgentURL, "application/json",
                         strings.NewReader(string(reqBody)))

    respBody, _ := io.ReadAll(resp.Body)
    return string(respBody), nil
}
```

Doc Agent

```go
// Specialized tools for documentation
var DocTools = []ToolDefinition{
    SearchGoDocumentationDefinition,
}

// DocAgent reads and writes to the network
docAgent := NewAgent(
    "docAgent",
    client,
    DocTools,
    ReadFromNetwork,
    WriteToNetwork,
)
```

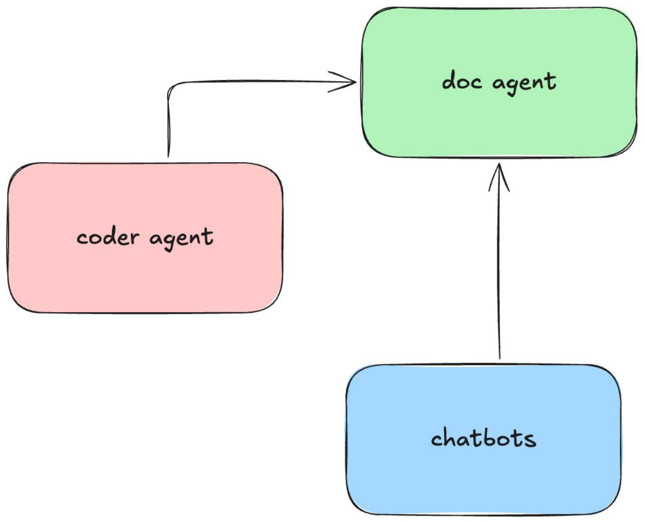
```go
// Coder agent calls Doc agent over the network
func InvokeDocumentationAgent(input json.RawMessage) (string, error) {
    var input InvokeDocumentationAgentInput
    json.Unmarshal(input, &input)

    docAgentURL := os.Getenv("DOC_AGENT_URL") // Service discovery!
    payload := map[string]string{"query": input.Query}
    reqBody, _ := json.Marshal(payload)

    resp, _ := http.Post(docAgentURL, "application/json",
                        strings.NewReader(string(reqBody)))

    respBody, _ := io.ReadAll(resp.Body)
    return string(respBody), nil
}
```

Demo our multi-agent system

Demo: Is that enough to make our agents talk to each other?

Deploying our multi-agent system

You could deploy our entire system as a binary, however not every agent might have the same execution patterns.



Containerization

Containers can help solve this challenge! Go lends itself really well to containerization because:

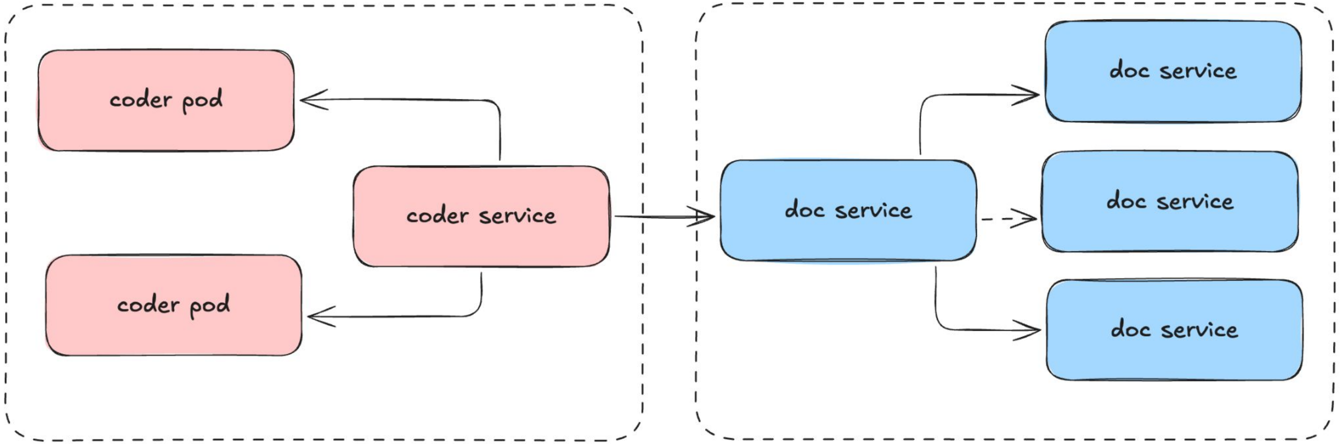
1. Statically linked, compiled binary
2. Cross platform
3. Cloud-native ecosystem

Containerization

```
FROM golang:1.23-alpine AS builder

WORKDIR /app
COPY go.mod go.sum ./
RUN go mod download
COPY . .
RUN CGO_ENABLED=0 GOOS=linux go build -o coder-agent .

FROM alpine:latest
WORKDIR /root/
COPY --from=builder /app/coder-agent .
ENV AGENT_TYPE=coder
ENV PORT=8080
EXPOSE 8080

CMD ["./coder-agent"]
```

Orchestration

1. To orchestrate our containers in production we can use Kubernetes.
2. This allows us to build upon a wealth of experience from existing production patterns.



Conclusion

Key idea: Building agents in Golang is easy, and allows you to immediately leverage existing production patterns.

1. Lots of open-source AI agent ecosystem libraries such as LangChainGo, Weaviate, and Ollama, all written in Go.
2. This is a great time to contribute!

Thanks!