

Agent Usage Documentation

Agent is the core execution unit of the tRPC-Agent-Go framework, responsible for processing user input and generating corresponding responses. Each Agent implements a unified interface, supporting streaming output and callback mechanisms.

The framework provides multiple types of Agents, including LLMAgent, ChainAgent, ParallelAgent, CycleAgent, and GraphAgent. This document focuses on LLMAgent. For detailed information about other Agent types and multi-Agent systems, please refer to [Multi-Agent](#).

Quick Start

Recommended Usage: Runner

We strongly recommend using Runner to execute Agents instead of directly calling Agent interfaces. Runner provides a more user-friendly interface, integrating services like Session and Memory, making usage much simpler.

 **Learn More:** For detailed usage methods, please refer to [Runner](#)

This example uses OpenAI's GPT-4o-mini model. Before starting, please ensure you have prepared the corresponding `OPENAI_API_KEY` and exported it through environment variables:

```
1 export OPENAI_API_KEY="your_api_key"
```

Additionally, the framework supports OpenAI API-compatible models, which can be configured through environment variables:

```
1 export OPENAI_BASE_URL="your_api_base_url"
2 export OPENAI_API_KEY="your_api_key"
```

Creating Model Instance

First, you need to create a model instance. Here we use OpenAI's GPT-4o-mini model:

```
1
2
3
4
5
6
```

```
import "trpc.group/trpc-go/trpc-agent-go/model/openai"

modelName := flag.String("model", "gpt-4o-mini", "Name of the model to
use")
flag.Parse()
// Create OpenAI model instance.
modelInstance := openai.New(*modelName, openai.Options{})
```

Configuring Generation Parameters

Set the model's generation parameters, including maximum tokens, temperature, and whether to use streaming output:

```
1 import "trpc.group/trpc-go/trpc-agent-go/model"
2
3 maxTokens := 1000
4 temperature := 0.7
5 genConfig := model.GenerationConfig{
6     MaxTokens: &maxTokens, // Maximum number of tokens to generate.
7     Temperature: &temperature, // Temperature parameter, controls output
8     randomness.
9     Stream:      true, // Enable streaming output.
10 }
```

Creating LLMAgent

Use the model instance and configuration to create an LLMAgent, while setting the Agent's Description and Instruction.

Description is used to describe the basic functionality and characteristics of the Agent, while Instruction defines the specific instructions and behavioral guidelines that the Agent should follow when executing tasks.

```
1 import "trpc.group/trpc-go/trpc-agent-go/agent/llmagent"
2
3 llmAgent := llmagent.New(
4     "demo-agent", // Agent name.
5     llmagent.WithModel(modelInstance), // Set model.
6     llmagent.WithDescription("A helpful AI assistant for
7 demonstrations"), // Set description.
8     llmagent.WithInstruction("Be helpful, concise, and informative in
9 your responses"), // Set instruction.
10    llmagent.WithGenerationConfig(genConfig),
11    // Set generation parameters.
12    // Set the filter mode for messages passed to the model. The final
13    messages passed to the model must satisfy both
14    WithMessageTimelineFilterMode and WithMessageBranchFilterMode conditions.
15    // Timeline dimension filter conditions
16    // Default: llmagent.TimelineFilterAll
17    // Optional values:
```

```

18 // - llmagent.TimelineFilterAll: Includes historical messages as
19 well as messages generated in the current request
20 // - llmagent.TimelineFilterCurrentRequest: Only includes messages
21 generated in the current request
22 // - llmagent.TimelineFilterCurrentInvocation: Only includes
23 messages generated in the current invocation context
24 llmagent.WithMessageTimelineFilterMode(llmagent.TimelineFilterAll),
25
    // Branch dimension filter conditions
    // Default: llmagent.BranchFilterModePrefix
    // Optional values:
    // - llmagent.BranchFilterModeAll: Includes messages from all
    agents. Use this when the current agent interacts with the model and
    needs to synchronize all valid content messages generated by all agents
    to the model.
    // - llmagent.BranchFilterModePrefix: Filters messages by prefix
    matching Event.FilterKey with Invocation.eventFilterKey. Use this when
    you want to pass messages generated by the current agent and related
    upstream/downstream agents to the model.
    // - llmagent.BranchFilterModeExact: Filters messages where
    Event.FilterKey == Invocation.eventFilterKey. Use this when the current
    agent interacts with the model and only needs to use messages generated
    by the current agent.
    llmagent.WithMessageBranchFilterMode(llmagent.BranchFilterModeAll),
    )

```

Placeholder Variables (Session State Injection)

LLMAgent automatically injects session state into `Instruction` and the optional `SystemPrompt` via placeholder variables. Supported patterns:

- `{key}` : Replaced with the string value corresponding to the key `key` in the session state (write via `invocation.Session.SetState("key", ...)` or `SessionService`)
- `{key?}` : Optional; if missing, replaced with an empty string
- `{user:subkey}` / `{app:subkey}` / `{temp:subkey}` : Use user/app/temp scoped keys (session services merge app/user state into session with these prefixes)
- `{invocation:subkey}` : Replaces with the value of `fmt.Sprintf("%+v", invocation.state["subkey"])`. (The state can be set via `invocation.SetState(k, v)`)

Notes:

- If a non-optional key is not found, the original `{key}` is preserved (helps the LLM notice missing context)
- Values are read from session state (Runner + `SessionService` set/merge this automatically)

Example:

```

1  llm := llmagent.New(
2      "research-agent",
3      llmagent.WithModel(modelInstance),
4      llmagent.WithInstruction(
5          "You are a research assistant. Focus: {research_topics}. " +
6          "User interests: {user:topics?}. App banner: {app:banner?}.",
7      ),
8  )
9
10 inv := agent.NewInvocaction()
11 inv.SetState("case", "case-1")
12
13 // Initialize session state (Runner + SessionService)
14 _ = sessionService.UpdateUserState(ctx, session.UserKey{AppName: app,
15 UserID: user}, session.StateMap{
16     "topics": []byte("quantum computing, cryptography"),
17 })
18 _ = sessionService.UpdateAppState(ctx, app, session.StateMap{
19     "banner": []byte("Research Mode"),
20 })
21 // Unprefixed keys live directly in session.State
22 _, _ = sessionService.CreateSession(ctx, session.Key{AppName: app,
23 UserID: user, SessionID: sid}, session.StateMap{
24     "research_topics": []byte("AI, ML, DL"),
25 })

```

See also:

- Examples: [examples/placeholder](#), [examples/outputkey](#)
- Session API: [docs/mkdocs/en/session.md](#)

Using Runner to Execute Agent

Use Runner to execute the Agent, which is the recommended usage:

```

1  import "trpc.group/trpc-go/trpc-agent-go/runner"
2
3  // Create Runner.
4  runner := runner.NewRunner("demo-app", llmAgent)
5
6  // Send message directly without creating complex Invocation.
7  message := model.NewUserMessage("Hello! Can you tell me about yourself?")
8  eventChan, err := runner.Run(ctx, "user-001", "session-001", message)
9  if err != nil {
10     log.Fatalf("Failed to execute Agent: %v", err)
11 }

```

Message Visibility Options

The Agent can dynamically manage the visibility of messages generated by other Agents and historical session messages based on different scenarios. This is configurable through

relevant options. When interacting with the model, only the visible content is passed as input.

TIPS: - Messages from different sessionIDs are never visible to each other under any circumstances. The following control strategies only apply to messages sharing the same sessionID. - Invocation.Message always visible regardless of the configuration. - When the option is not configured, the default value is FullContext.

Config: - `llmagent.WithMessageFilterMode(MessageFilterMode)` :- `FullContext` : Includes historical messages and messages generated in the current request, filtered by prefix matching with the filterKey. - `RequestContext` : Only includes messages generated in the current request, filtered by prefix matching with the filterKey. - `IsolatedRequest` : Only includes messages generated in the current request, filtered by exact matching with the filterKey. - `IsolatedInvocation` : Only includes messages generated in the current Invocation context, filtered by exact matching with the filterKey.

Recommended Usage Examples (These examples are simplified configurations based on advanced usage):

```

1  taskagentA := llmagent.New(
2      "coordinator",
3      llmagent.WithModel(modelInstance),
4      // Makes all messages generated by taskagentA and taskagentB visible
5      (including historical session messages under the same sessionID)
6      llmagent.WithMessageFilterMode(llmagent.FullContext),
7      // Makes all messages generated during the current runner.Run of
8      taskagentA and taskagentB visible (excluding historical session messages)
9      llmagent.WithMessageFilterMode(llmagent.RequestContext),
10     // Only makes messages generated during the current runner.Run of
11     taskagentA visible (excluding its own historical session messages)
12     llmagent.WithMessageFilterMode(llmagent.IsolatedRequest),
13     // Agent execution order: taskagentA-invocation1 -> taskagentB-
14     invocation2 -> taskagentA-invocation3 (current execution phase)
15     // Only makes messages generated during the current taskagentA-
16     invocation3 phase visible (excluding its own historical session messages
17     and messages generated during taskagentA-invocation1)
18     llmagent.WithMessageFilterMode(llmagent.IsolatedInvocation),
19 )
20
21 taskagentB := llmagent.New(
22     "coordinator",
23     llmagent.WithModel(modelInstance),
24     // Makes all messages generated by taskagentA and taskagentB visible
25     (including historical session messages under the same sessionID)
26     llmagent.WithMessageFilterMode(llmagent.FullContext),
27     // Makes all messages generated during the current runner.Run of
28     taskagentA and taskagentB visible (excluding historical session messages)
29     llmagent.WithMessageFilterMode(llmagent.RequestContext),
30     // Only makes messages generated during the current runner.Run of
31     taskagentB visible (excluding its own historical session messages)
32     llmagent.WithMessageFilterMode(llmagent.IsolatedRequest),
33     // Agent execution order: taskagentA-invocation1 -> taskagentB-
34     invocation2 -> taskagentA-invocation3 -> taskagentB-invocation4 (current

```

```

35 execution phase)
36 // Only makes messages generated during the current taskagentB-
37 invocation4 phase visible (excluding its own historical session messages
38 and messages generated during taskagentB-invocation2)
39 llmagent.WithMessageFilterMode(llmagent.IsolatedInvocation),
40 )
41
42 // Cyclically execute taskagentA and taskagentB
43 cycleAgent := cycleagent.New(
44     "coordinator",
45     llmagent.WithModel(modelInstance),
46     llmagent.WithSubAgents([]agent.Agent{taskagentA, taskagentB}),
47     llmagent.WithMessageFilterMode(llmagent.FullContext)
48 )
49
50 // Create Runner
51 runner := runner.NewRunner("demo-app", cycleAgent)
52
53 // Send message directly without creating complex Invocation
54 message := model.NewUserMessage("Hello! Can you tell me about yourself?")
55 eventChan, err := runner.Run(ctx, "user-001", "session-001", message)
56 if err != nil {
57     log.Fatalf("Failed to run Agent: %v", err)
58 }

```

Advanced Usage Examples: You can independently control the visibility of historical messages and messages generated by other Agents for the current agent using `WithMessageTimelineFilterMode` and `WithMessageBranchFilterMode`. When the current agent interacts with the model, only messages satisfying both conditions are input to the model. (invocation.Messages is always visible in any scenario.)

- `WithMessageTimelineFilterMode`: Controls visibility from a temporal dimension
 - `TimelineFilterAll`: Includes historical messages and messages generated in the current request.
 - `TimelineFilterCurrentRequest`: Only includes messages generated in the current request (one runner.Run counts as one request).
 - `TimelineFilterCurrentInvocation`: Only includes messages generated in the current invocation context.
- `WithMessageBranchFilterMode`: Controls visibility from a branch dimension (used to manage visibility of messages generated by other agents).
 - `BranchFilterModePrefix`: Uses prefix matching between `Event.FilterKey` and `Invocation.eventFilterKey`.
 - `BranchFilterModeAll`: Includes messages from all agents.
 - `BranchFilterModeExact`: Only includes messages generated by the current agent.

```

1 llmAgent := llmagent.New(
2     "demo-agent", // Agent name
3     llmagent.WithModel(modelInstance), // Set the model
4     llmagent.WithDescription("A helpful AI assistant for
5     demonstrations"), // Set description
6     llmagent.WithInstruction("Be helpful, concise, and informative in
7     your responses"), // Set instruction
8     llmagent.WithGenerationConfig(genConfig),
9     // Set generation parameters
10
11     // Set the message filtering mode for input to the model. The final
12     messages passed to the model must satisfy both

```

```
13 WithMessageTimelineFilterMode and WithMessageBranchFilterMode conditions.
14     // Temporal dimension filtering condition
15     // Default: llmagent.TimelineFilterAll
16     // Options:
17     // - llmagent.TimelineFilterAll: Includes historical messages and
18 messages generated in the current request.
19     // - llmagent.TimelineFilterCurrentRequest: Only includes messages
20 generated in the current request.
21     // - llmagent.TimelineFilterCurrentInvocation: Only includes
22 messages generated in the current invocation context.
23     llmagent.WithMessageTimelineFilterMode(llmagent.TimelineFilterAll),
    // Branch dimension filtering condition
    // Default: llmagent.BranchFilterModePrefix
    // Options:
    // - llmagent.BranchFilterModeAll: Includes messages from all
agents. Use this when the current agent needs to sync valid content
messages generated by all agents to the model during interaction.
    // - llmagent.BranchFilterModePrefix: Filters messages by prefix
matching Event.FilterKey with Invocation.eventFilterKey. Use this when
you want to pass messages generated by the current agent and related
upstream/downstream agents to the model.
    // - llmagent.BranchFilterModeExact: Filters messages where
Event.FilterKey exactly matches Invocation.eventFilterKey. Use this when
the current agent only needs to use messages generated by itself during
model interaction.
    llmagent.WithMessageBranchFilterMode(llmagent.BranchFilterModeAll),
)
```

Reasoning Content Mode (DeepSeek Thinking Mode)

When using models with thinking/reasoning capabilities (such as DeepSeek), the model outputs both `reasoning_content` (thinking chain) and `content` (final answer). According to [DeepSeek API documentation](#), in multi-turn conversations, you should not send the previous turn's `reasoning_content` to the model.

LLMAgent provides `WithReasoningContentMode` to control how `reasoning_content` is handled in conversation history:

Available Modes:

Mode	Constant	Description
Discard Previous Turns	<code>ReasoningContentModeDiscardPreviousTurns</code>	Discard <code>reasoning_content</code> from previous request turns, keep for current request. (Default, recommended)
Keep All	<code>ReasoningContentModeKeepAll</code>	Keep all <code>reasoning_content</code> in history (for debugging).

Mode	Constant	Description
Discard All	<code>ReasoningContentModeDiscardAll</code>	Discard all <code>reasoning_content</code> from history for maximum bandwidth savings.

Usage Example:

```
1 // Recommended configuration for DeepSeek models with thinking mode.
2 agent := llmagent.New(
3     "deepseek-agent",
4     llmagent.WithModel(deepseekModel),
5     llmagent.WithInstruction("You are a helpful assistant."),
6     // Discard reasoning_content from previous turns (recommended for DeepSeek)
7     llmagent.WithReasoningContentMode(llmagent.ReasoningContentModeDiscardPreviousTurns)
8 )
```

How It Works:

- `keep_all`: All `reasoning_content` is preserved in session history. Use this if you need to retain thinking chains for debugging or analysis.
- `discard_previous_turns`: When building the message list for a new request, `reasoning_content` from messages belonging to previous requests is cleared. Messages within the current request (e.g., during tool call loops) retain their `reasoning_content`. This follows DeepSeek's recommendation.
- `discard_all`: All `reasoning_content` is stripped from historical messages before sending to the model.

Note: This option only affects how historical messages are processed before sending to the model. The current response's `reasoning_content` is always captured and stored in session events.

Delegation Visibility Options

When building multi-Agent systems (task delegation between Agents), LLMAgent provides a unified fallback option for delegation events. Transfer events always include announcement text and are tagged `transfer` so UIs (User Interfaces) can filter them if desired.

- `llmagent.WithDefaultTransferMessage(string)`
 - Configure the default message used when a model calls a SubAgent without a `message`.

- Pass an empty string to disable injecting a default message; pass a non-empty string to enable and override it.

Usage example:

```

1 coordinator := llmagent.New(
2     "coordinator",
3     llmagent.WithModel(modelInstance),
4     llmagent.WithSubAgents([]agent.Agent{mathAgent, weatherAgent}),
5     // Transfer announcement events are always emitted (tagged `transfer`).
6     Filter in the UI if needed.
7     // Customize the default message when the model omits it (empty string
8     disables)
9     llmagent.WithDefaultTransferMessage("Handing off to the specialist"),
10 )

```

Notes:

- These options do not change the actual handoff logic; they only affect user-visible texts or whether a fallback `message` is injected.
- Transfer announcements are emitted as Events with `Response.Object == "agent.transfer"`. If your UI should not display system-level notices, filter this object type at the renderer/service layer.

Handling Event Stream

The `eventChan` returned by `runner.Run()` is an event channel. The Agent continuously sends Event objects to this channel during execution.

Each Event contains execution state information at a specific moment: LLM-generated content, tool call requests and results, error messages, etc. By iterating through the event channel, you can get real-time execution progress (see [Event](#) section below for details).

Receive execution results through the event channel:

```

1 // 1. Get event channel (returns immediately, starts async execution)
2 eventChan, err := runner.Run(ctx, userID, sessionID, message)
3 if err != nil {
4     log.Fatalf("Failed to start: %v", err)
5 }
6
7 // 2. Handle event stream (receive execution results in real-time)
8 for event := range eventChan {
9     // Check for errors
10    if event.Error != nil {
11        log.Printf("Execution error: %s", event.Error.Message)
12        continue
13    }
14
15    // Handle response content

```

```

16     if len(event.Response.Choices) > 0 {
17         choice := event.Response.Choices[0]
18
19         // Streaming content (real-time display)
20         if choice.Delta.Content != "" {
21             fmt.Print(choice.Delta.Content)
22         }
23
24         // Tool call information
25         for _, toolCall := range choice.Message.ToolCalls {
26             fmt.Printf("Calling tool: %s\n", toolCall.Function.Name)
27         }
28     }
29
30     // Check if completed (note: should not break on tool call
31     completion)
32     if event.IsFinalResponse() {
33         fmt.Println()
34         break
35     }
36 }

```

The complete code for this example can be found at [examples/runner](#)

Why is Runner recommended?

1. **Simpler Interface:** No need to create complex Invocation objects
2. **Integrated Services:** Automatically integrates Session, Memory and other services
3. **Better Management:** Unified management of Agent execution flow
4. **Production Ready:** Suitable for production environment use

💡 **Tip:** Want to learn more about Runner's detailed usage and advanced features? Please check [Runner](#)

Advanced Usage: Direct Agent Usage

If you need more fine-grained control, you can also use the Agent interface directly, but this requires creating Invocation objects:

Core Concepts

Invocation (Advanced Usage)

Invocation is the context object for Agent execution flow, containing all information needed for a single call. **Note: This is advanced usage, we recommend using Runner to simplify operations.**

```

1  import "trpc.group/trpc-go/trpc-agent-go/agent"
2

```

```

3 // Create Invocation object (advanced usage).
4 invocation := agent.NewInvocation(
5     agent.WithAgentName(agent),
6     // Agent.
7     agent.WithInvocationMessage(model.NewUserMessage("Hello! Can you tell
8 me about yourself?")), // User message.
9     agent.WithInvocationSession(&session.Session{ID: "session-001"}),
10    // session object.
11    agent.WithInvocationEndInvocation(false),
12    // Whether to end invocation.
13    agent.WithInvocationModel(modelInstance),
14    // Model to use.
15 )
16
17 // Call Agent directly (advanced usage).
ctx := context.Background()
eventChan, err := llmAgent.Run(ctx, invocation)
if err != nil {
    log.Fatalf("Failed to execute Agent: %v", err)
}

```

When to use direct calls?

- Need complete control over execution flow
- Custom Session and Memory management
- Implement special invocation logic
- Debugging and testing scenarios

```

1 // Invocation is the context object for Agent execution flow, containing
2 // all information needed for a single call.
3 type Invocation struct {
4     // Agent specifies the Agent instance to call.
5     Agent Agent
6     // AgentName identifies the name of the Agent instance to call.
7     AgentName string
8     // InvocationID provides a unique identifier for each call.
9     InvocationID string
10    // Branch is a branch identifier for hierarchical event filtering.
11    Branch string
12    // EndInvocation indicates whether to end the invocation.
13    EndInvocation bool
14
15    // Session maintains the context state of the conversation.
16    Session *session.Session
17    // Model specifies the model instance to use.
18    Model model.Model
19    // Message is the specific content sent by the user to the Agent.
20    Message model.Message
21    // RunOptions are option configurations for the Run call.
22    RunOptions RunOptions
23    // TransferInfo supports control transfer between Agents.
24    TransferInfo *TransferInfo
25
26    // Structured output configuration (optional).

```

```

27     StructuredOutput      *model.StructuredOutput
28     StructuredOutputType reflect.Type
29
30     // Services injected for this invocation.
31     MemoryService    memory.Service
32     ArtifactService  artifact.Service
33
34     // Internal signaling: notify when events are appended.
35     noticeChanMap map[string]chan any
36     noticeMu      *sync.Mutex
37
38     // Internal: event filter key and parent linkage for nested flows.
39     eventFilterKey string
40     parent        *Invocation
41
42     // Invocation-scoped state (lazy-init, thread-safe via stateMu).
43     state map[string]any
44     stateMu sync.RWMutex
45
46     // Optional per-invocation safety limits (usually set by LLMAgent).
47     MaxLLMCalls      int
48     MaxToolIterations int
49
50     // Internal counters used to enforce MaxLLMCalls / MaxToolIterations.
51     llmCallCount      int
52     toolIterationCount int
53 }

```

Invocation State

`Invocation` provides a general-purpose state storage mechanism for sharing data within the lifecycle of a single invocation. This is useful for callbacks, middleware, or any scenario that requires storing temporary data at the invocation level.

Core Methods:

```

1 // Set a state value
2 inv.SetState(key string, value any)
3
4 // Get a state value
5 value, ok := inv.GetState(key string)
6
7 // Delete a state value
8 inv.DeleteState(key string)

```

Features:

- **Invocation-scoped:** State is automatically scoped to a single invocation
- **Thread-safe:** Built-in RWMutex protection for concurrent access
- **Lazy initialization:** Memory allocated only on first use
- **General-purpose:** Can be used for callbacks, middleware, custom logic, and more

Usage Example:

Version Requirement

The structured callback API (recommended) requires **trpc-agent-go** **>= 0.6.0**.

```

1 // Store data in BeforeAgentCallback
2 // Note: Structured callback API requires trpc-agent-go >= 0.6.0
3 callbacks := agent.NewCallbacks()
4 callbacks.RegisterBeforeAgent(func(ctx context.Context, args
5 *agent.BeforeAgentArgs) (*agent.BeforeAgentResult, error) {
6     args.Invocation.SetState("agent:start_time", time.Now())
7     args.Invocation.SetState("custom:request_id", "req-123")
8     return nil, nil
9 })
10
11 // Read data in AfterAgentCallback
12 callbacks.RegisterAfterAgent(func(ctx context.Context, args
13 *agent.AfterAgentArgs) (*agent.AfterAgentResult, error) {
14     if startTime, ok := args.Invocation.GetState("agent:start_time"); ok
15     {
16         duration := time.Since(startTime.(time.Time))
17         log.Printf("Execution took: %v", duration)
18         args.Invocation.DeleteState("agent:start_time")
19     }
20     return nil, nil
21 })

```

Recommended Key Naming Convention:

- Agent callbacks: "agent:xxx"
- Model callbacks: "model:xxx"
- Tool callbacks: "tool:toolName:xxx"
- Middleware: "middleware:xxx"
- Custom logic: "custom:xxx"

For detailed usage and more examples, please refer to [Callbacks](#).

Event

Event is the real-time feedback generated during Agent execution, reporting execution progress in real-time through Event streams.

Events mainly include the following types:

- Model conversation events
- Tool call and response events
- Agent transfer events

- Error events

```

1 // Event is the real-time feedback generated during Agent execution,
2 reporting execution progress in real-time through Event streams.
3 type Event struct {
4     // Response contains model response content, tool call results and
5     statistics.
6     *model.Response
7     // InvocationID is associated with a specific invocation.
8     InvocationID string `json:"invocationId"`
9     // Author is the source of the event, such as Agent or tool.
10    Author string `json:"author"`
11    // ID is the unique identifier of the event.
12    ID string `json:"id"`
13    // Timestamp records the time when the event occurred.
14    Timestamp time.Time `json:"timestamp"`
15    // Branch is a branch identifier for hierarchical event filtering.
16    Branch string `json:"branch,omitempty"`
17    // RequiresCompletion identifies whether this event requires a
18    completion signal.
19    RequiresCompletion bool `json:"requiresCompletion,omitempty"`
20    // LongRunningToolIDs is a set of IDs for long-running function
21    calls. Agent clients can understand which function calls are long-running
22    through this field, only valid for function call events.
23    LongRunningToolIDs map[string]struct{}
24    `json:"longRunningToolIDs,omitempty"`
25 }

```

The streaming nature of Events allows you to see the Agent's working process in real-time, just like having a natural conversation with a real person. You only need to iterate through the Event stream, check the content and status of each Event, and you can completely handle the Agent's execution results.

Agent Interface

The Agent interface defines the core behaviors that all Agents must implement. This interface allows you to uniformly use different types of Agents while supporting tool calls and sub-Agent management.

```

1 type Agent interface {
2     // Run receives execution context and invocation information, returns
3     an event channel. Through this channel, you can receive Agent execution
4     progress and results in real-time.
5     Run(ctx context.Context, invocation *Invocation) (<-chan
6     *event.Event, error)
7     // Tools returns the list of tools that this Agent can access and
8     execute.
9     Tools() []tool.Tool
10    // Info method provides basic information about the Agent, including
11    name and description, for easy identification and management.
12    Info() Info
13    // SubAgents returns the list of sub-Agents available to this Agent.

```

```
// SubAgents and FindSubAgent methods support collaboration between
Agents. An Agent can delegate tasks to other Agents, building complex
multi-Agent systems.
SubAgents() []Agent
// FindSubAgent finds sub-Agent by name.
FindSubAgent(name string) Agent
}
```

The framework provides multiple types of Agent implementations, including LLMAgent, ChainAgent, ParallelAgent, CycleAgent, and GraphAgent. For detailed information about different types of Agents and multi-Agent systems, please refer to [Multi-Agent](#).

Callbacks

Callbacks provide a rich callback mechanism that allows you to inject custom logic at key points during Agent execution.

Version Requirement

The structured callback API (recommended) requires **trpc-agent-go** **>= 0.6.0**.

Callback Types

The framework provides three types of callbacks:

Agent Callbacks: Triggered before and after Agent execution

```
1 // Create callbacks using agent.NewCallbacks()
2 callbacks := agent.NewCallbacks()
```

Model Callbacks: Triggered before and after model calls

```
1 // Create callbacks using model.NewCallbacks()
2 callbacks := model.NewCallbacks()
```

Tool Callbacks: Triggered before and after tool calls

```
1 // Create callbacks using tool.NewCallbacks()
2 callbacks := tool.NewCallbacks()
```

Usage Example

```
1 // Create Agent callbacks (using structured API)
2 // Note: Structured callback API requires trpc-agent-go >= 0.6.0
3 callbacks := agent.NewCallbacks()
4 callbacks.RegisterBeforeAgent(func(ctx context.Context, args
5 *agent.BeforeAgentArgs) (*agent.BeforeAgentResult, error) {
```

```
6     log.Printf("Agent %s started execution", args.Invocation.AgentName)
7     return nil, nil
8 })
9 callbacks.RegisterAfterAgent(func(ctx context.Context, args
10 *agent.AfterAgentArgs) (*agent.AfterAgentResult, error) {
11     if args.Error != nil {
12         log.Printf("Agent %s execution error: %v",
13 args.Invocation.AgentName, args.Error)
14     } else {
15         log.Printf("Agent %s execution completed",
16 args.Invocation.AgentName)
17     }
18     return nil, nil
19 })

// Use callbacks in llmAgent
llmagent := llmagent.New("llmagent",
llmagent.WithAgentCallbacks(callbacks))
```

The callback mechanism allows you to precisely control the Agent's execution process and implement more complex business logic.

Structured Output

Structured output ensures that agent responses conform to a predefined format, making them easier to parse and process programmatically. The framework provides multiple methods for structured output, each suited for different use cases.

Comparison of Structured Output Methods

Feature	WithStructuredOutputJSONSchema	WithStructuredOutputJSON	WithOut
Tool Usage	✅ Allowed	✅ Allowed	❌ Disa
Schema Type	User-provided JSON Schema	Auto-generated from Go struct	User-pro JSON S
Output Type	Untyped (map/interface{})	Typed (Go struct)	Untyped (map/in
Schema Validation	✅ By LLM	✅ By LLM	✅ By L
Data Location	Event.StructuredOutput	Event.StructuredOutput	Model r content

Feature	WithStructuredOutputJSONSchema	WithStructuredOutputJSON	Without
Primary Use Case	Flexible schema with tools	Type-safe structured output	Simple responses

WithStructuredOutputJSONSchema

Provides a user-defined JSON schema for structured output while **allowing tool usage**. This is the most flexible option for agents that need both structured output and tool capabilities.

Example:

```
1  schema := map[string]any{
2      "type": "object",
3      "properties": map[string]any{
4          "name": map[string]any{
5              "type": "string",
6              "description": "Product name",
7          },
8          "price": map[string]any{
9              "type": "number",
10             "minimum": 0,
11         },
12         "category": map[string]any{
13             "type": "string",
14             "enum": []string{"electronics", "clothing", "food"},
15         },
16     },
17     "required": []string{"name", "price"},
18 }
19
20 agent := llmagent.New(
21     "shopping-agent",
22     llmagent.WithModel(modelInstance),
23     llmagent.WithStructuredOutputJSONSchema(
24         "shopping_output", // Name
25         schema,           // JSON schema
26         true,              // Strict mode
27         "Product information", // Description
28     ),
29     llmagent.WithTools([]tool.Tool{searchTool, calculatorTool}), // Tools
30     // are allowed!
31 )
32
33 // Access untyped output from events
34 for event := range eventCh {
35     if event.StructuredOutput != nil {
36         data := event.StructuredOutput.(map[string]any)
37         name := data["name"].(string)
38         price := data["price"].(float64)
```

```

39         fmt.Printf("Product: %s, Price: $%.2f\n", name, price)
40     }
    }

```

Best for: - Complex agents requiring both structured output and tool usage - Working with external JSON schemas (from APIs, databases, config files) - Prototyping with dynamic schemas - Gradual typing scenarios

WithStructuredOutputJSON

Auto-generates JSON schema from a Go struct and returns typed output. Provides compile-time type safety.

Example:

```

1  type ProductInfo struct {
2      Name      string `json:"name"`
3      Price     float64 `json:"price"`
4      Category  string `json:"category"`
5  }
6
7  agent := llmagent.New(
8      "shopping-agent",
9      llmagent.WithModel(modelInstance),
10     llmagent.WithStructuredOutputJSON(
11         new(ProductInfo),           // Auto-generates schema
12         true,                       // Strict mode
13         "Product information",      // Description
14     ),
15     llmagent.WithTools([]tool.Tool{searchTool}), // Tools are allowed
16 )
17
18 // Access typed output from events
19 for event := range eventCh {
20     if event.StructuredOutput != nil {
21         product := event.StructuredOutput.(*ProductInfo)
22         fmt.Printf("Product: %s, Price: $%.2f\n", product.Name,
23             product.Price)
24     }
25 }

```

Best for: - Type-safe applications with well-defined Go structs - Clean code integration - Compile-time type checking

WithOutputSchema (Legacy)

Similar to `WithStructuredOutputJSONSchema` but **disables all tools**. This is a legacy method kept for backward compatibility.

Example:

```

1  agent := llmagent.New(
2      "weather-agent",
3      llmagent.WithModel(modelInstance),
4      llmagent.WithOutputSchema(weatherSchema),
5      // llmagent.WithTools(...) // ❌ Tools are disabled!
6  )

```

Limitations: - ❌ Cannot use tools, function calling, or RAG - ❌ Response in model content (needs parsing)

Migration Tip: If you need tool capabilities, migrate to `WithStructuredOutputJSONSchema` :

```

1  // Old: Tools disabled
2  agent := llmagent.New(
3      "agent",
4      llmagent.WithOutputSchema(schema),
5  )
6
7  // New: Tools enabled
8  agent := llmagent.New(
9      "agent",
10     llmagent.WithStructuredOutputJSONSchema(
11         "agent_output", // Name
12         schema,         // JSON schema
13         true,           // Strict mode
14         "Agent output", // Description
15     ),
16     llmagent.WithTools([]tool.Tool{myTool1, myTool2}), // ✅ Now works!
17 )

```

WithOutputKey

Stores agent output in session state under a specific key, useful for agent workflows where output needs to be accessed by downstream agents.

Example:

```

1  researchAgent := llmagent.New(
2      "researcher",
3      llmagent.WithModel(modelInstance),
4      llmagent.WithOutputKey("research_findings"),
5  )
6
7  writerAgent := llmagent.New(
8      "writer",
9      llmagent.WithModel(modelInstance),
10     llmagent.WithInstruction("Based on the research: {research_findings},
11     write a summary."),
12 )
13
14 // Chain agents using session state
15

```

```
16 chain := chainagent.New("pipeline",
17 chainagent.WithSubAgents([]agent.Agent{
    researchAgent,
    writerAgent,
}))
```

Best for: - Multi-agent workflows with data passing - Session state management - Placeholder variable access in downstream agents

Choosing the Right Method

Scenario	Recommended Method
Need tools + structured output	WithStructuredOutputJSONSchema or WithStructuredOutputJSON
Type safety is critical	WithStructuredOutputJSON
Working with external schemas	WithStructuredOutputJSONSchema
Simple structured responses (no tools)	WithOutputSchema
Multi-agent workflows	WithOutputKey
Rapid prototyping	WithStructuredOutputJSONSchema

Examples: - `examples/structuredoutput/` - Demonstrates `WithStructuredOutputJSON` (typed) - `examples/outputschema/` - Demonstrates `WithOutputSchema` (legacy) - `examples/outputkey/` - Demonstrates `WithOutputKey` (session state)

Advanced Usage

The framework provides advanced features like Runner, Session, and Memory for building more complex Agent systems.

Runner is the recommended usage, responsible for managing Agent execution flow, connecting Session/Memory Service capabilities, and providing a more user-friendly interface.

Session Service is used to manage session state, supporting conversation history and context maintenance.

Memory Service is used to record user preference information, supporting personalized experiences.

Recommended Reading Order:

1. [Runner](#) - Learn the recommended usage
2. [Session](#) - Understand session management
3. [Multi-Agent](#) - Learn multi-Agent systems

Runtime Instruction Updates

You can update an Agent's behavior-defining text at runtime without rebuilding or restarting the Agent.

What changes dynamically

- Instruction: the behavior guideline appended to the system message.
- Global Instruction (system prompt): the system-level preface prepended to the request.

Both can be updated on an existing `LLMagent` instance and take effect on subsequent model requests.

Example

```

1  import (
2      "context"
3
4      "trpc.group/trpc-go/trpc-agent-go/agent/llmagent"
5      "trpc.group/trpc-go/trpc-agent-go/model"
6      "trpc.group/trpc-go/trpc-agent-go/model/openai"
7      "trpc.group/trpc-go/trpc-agent-go/runner"
8  )
9
10 // 1) Build model and agent once at startup.
11 mdl := openai.New("gpt-4o-mini", openai.Options{})
12 llm := llmagent.New(
13     "support-bot",
14     llmagent.WithModel(mdl),
15     llmagent.WithInstruction("Be helpful and concise."),
16 )
17 run := runner.NewRunner("my-app", llm)
18
19 // 2) Later, change behavior at runtime (e.g., user updates prompt in
20 UI).
21 llm.SetInstruction("Translate all user inputs to French.")
22 llm.SetGlobalInstruction("System: Safety first. No PII leakage.")
23
24 // 3) Subsequent runs use the new instructions.
25 msg := model.NewUserMessage("Where is the nearest museum?")
26 ch, err := run.Run(context.Background(), "u1", "s1", msg)

```

```
_ = ch; _ = err
```

Notes

- Thread-safe: the setters are concurrency-safe and can be called while the service is handling requests.
- Mid-turn behavior: if an Agent's current turn triggers more than one model request (e.g., due to tool calls), updates may apply to subsequent requests in the same turn. If you need per-run stability, set or freeze the text at the start of the run.
- Model-specific prompts: if an Agent can switch models, use `llmagent.WithModelInstructions` / `llmagent.WithModelGlobalInstructions` (or the corresponding setters) to override prompts by `model.Info().Name`, falling back to the Agent defaults when no mapping exists.
- Per-session personalization: for per-user or per-session data, prefer placeholders in the instruction and session state injection (see the "Placeholder Variables" section above).

Model-specific Prompts

If a single Agent can switch between different models, you can define a different Instruction/system prompt for each model.

The key used for matching is the model name returned by `model.Info().Name`.

Example

```
1 import (
2     "trpc.group/trpc-go/trpc-agent-go/agent/llmagent"
3     "trpc.group/trpc-go/trpc-agent-go/model"
4     "trpc.group/trpc-go/trpc-agent-go/model/openai"
5 )
6
7 models := map[string]model.Model{
8     "gpt-4o-mini": openai.New("gpt-4o-mini"),
9     "gpt-4o":      openai.New("gpt-4o"),
10 }
11
12 llm := llmagent.New(
13     "support-bot",
14     llmagent.WithModels(models),
15     llmagent.WithModel(models["gpt-4o-mini"]), // Default model.
16
17     // Fallback prompts when no mapping exists.
18     llmagent.WithGlobalInstruction("System: You are a helpful
19 assistant."),
20     llmagent.WithInstruction("Start every answer with DEFAULT:"),
21
22     // Per-model prompt mapping.
23     llmagent.WithModelGlobalInstructions(map[string]string{
24         "gpt-4o-mini": "System: You are in FAST mode.",
```

```

25         "gpt-4o":      "System: You are in SMART mode.",
26     }},
27     llmagent.WithModelInstructions(map[string]string{
28         "gpt-4o-mini": "Start every answer with FAST:",
29         "gpt-4o":      "Start every answer with SMART:",
30     }},
    )

```

See also: `examples/model/promptmap`.

Alternative: Placeholder-Driven Dynamic System Prompts

If you'd rather not call setters, you can make the instruction itself a template and feed values via session state. The instruction processor replaces placeholders using session/app/user state on each turn.

Patterns

- Persistent per-user value: store under `user:*` and reference `{user:key}`.
- Persistent per-app value: store under `app:*` and reference `{app:key}`.
- Per-turn ephemeral value: write into the session's `temp:*` namespace and reference `{temp:key}` (not persisted).

Example: per-user dynamic instruction

```

1  import (
2      "context"
3
4      "trpc.group/trpc-go/trpc-agent-go/agent/llmagent"
5      "trpc.group/trpc-go/trpc-agent-go/runner"
6      "trpc.group/trpc-go/trpc-agent-go/session"
7      "trpc.group/trpc-go/trpc-agent-go/session/inmemory"
8  )
9
10 svc := inmemory.NewSessionService()
11 app, user, sid := "my-app", "u1", "s1"
12
13 // 1) Instruction template references a user-scoped key.
14 llm := llmagent.New(
15     "dyn-agent",
16     llmagent.WithInstruction("{user:system_prompt}"),
17 )
18 run := runner.NewRunner(app, llm, runner.WithSessionService(svc))
19
20 // 2) Update the user-scoped state when the user changes settings.
21 _ = svc.UpdateUserState(context.Background(), session.UserKey{AppName:
22 app, UserID: user}, session.StateMap{
23     "system_prompt": []byte("You are a helpful assistant. Always answer in
24 English."),
25 })
26
27 // 3) Runs now read the latest prompt via placeholder injection.

```

```
_, _ = run.Run(context.Background(), user, sid,
model.NewUserMessage("Hi!"))
```

Example: per-turn temp value via a before-agent callback

Version Requirement

The structured callback API (recommended) requires **trpc-agent-go >= 0.6.0**.

```
1 // Note: Structured callback API requires trpc-agent-go >= 0.6.0
2 callbacks := agent.NewCallbacks()
3 callbacks.RegisterBeforeAgent(func(ctx context.Context, args
4 *agent.BeforeAgentArgs) (*agent.BeforeAgentResult, error) {
5     if args.Invocation != nil && args.Invocation.Session != nil {
6         // Write a one-off instruction for this turn only
7         args.Invocation.Session.SetState("temp:sys", []byte("Translate to
8 French."))
9     }
10    return nil, nil
11 })
12
13 llm := llmagent.New(
14     "temp-agent",
15     llmagent.WithInstruction("{temp:sys}"),
16     llmagent.WithAgentCallbacks(callbacks), // requires trpc-agent-go >=
17     0.6.0
18 )
```

Caveats

- In-memory `UpdateUserState` intentionally forbids `temp:*` updates; write `temp:*` via `invocation.Session.SetState` (e.g., via a callback) when you need ephemeral, per-turn values.
- Placeholders are resolved at request time; changing the stored value updates behavior on the next model request without recreating the agent.

🕒 2026-01-08 03:39:14 🕒 2025-08-25 07:06:16