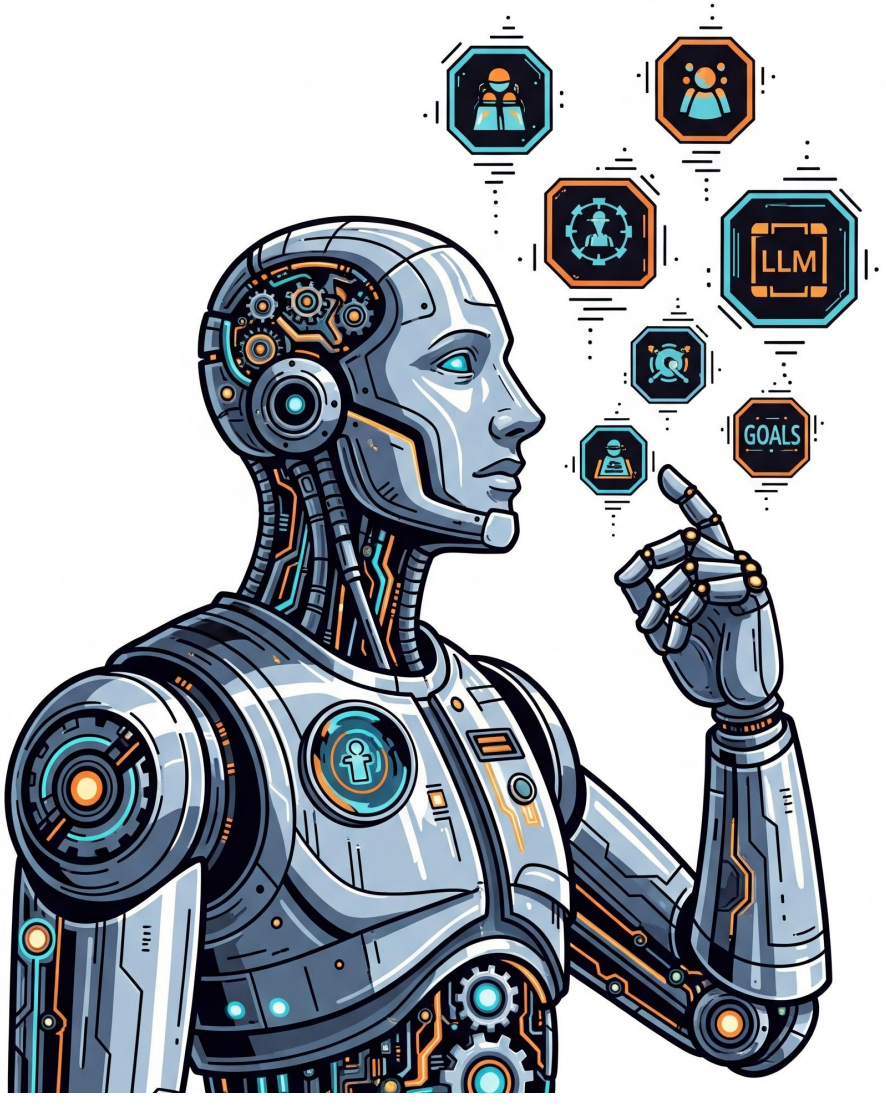
Beyond the Demo: Engineering Reliable LLM Agents with the 12-Factor Blueprint

Gurram Poorna Prudhvi

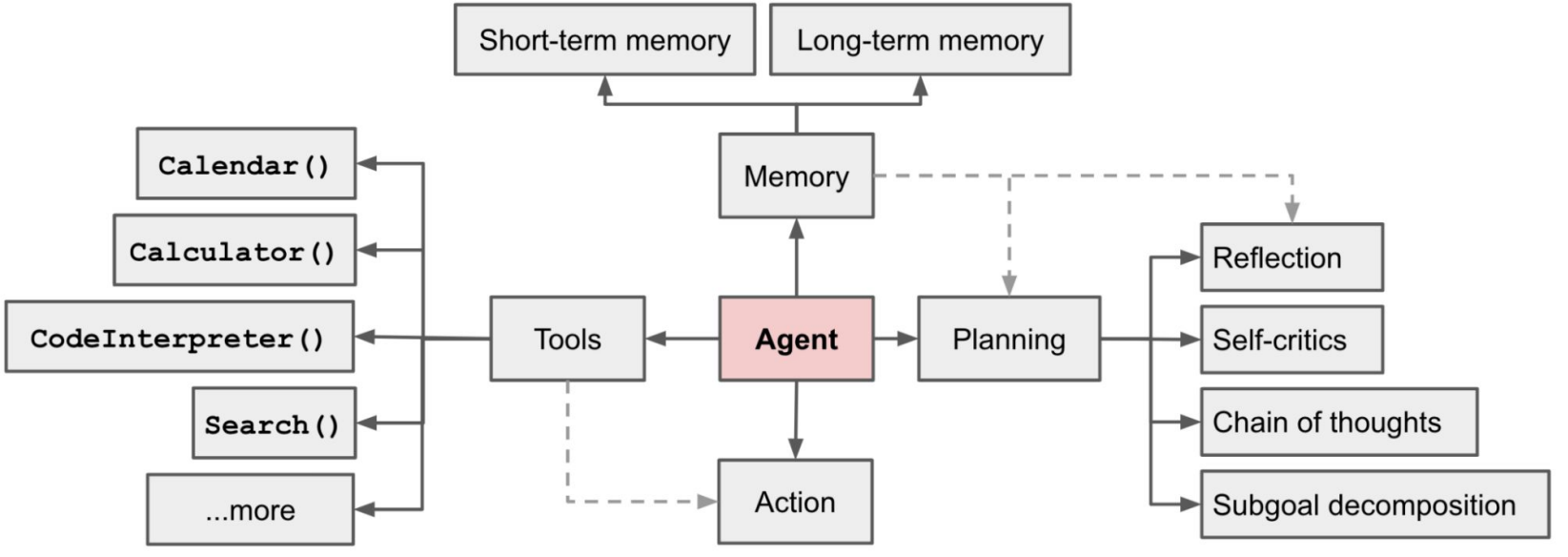
Lead software engineer @ epam

AI Agent

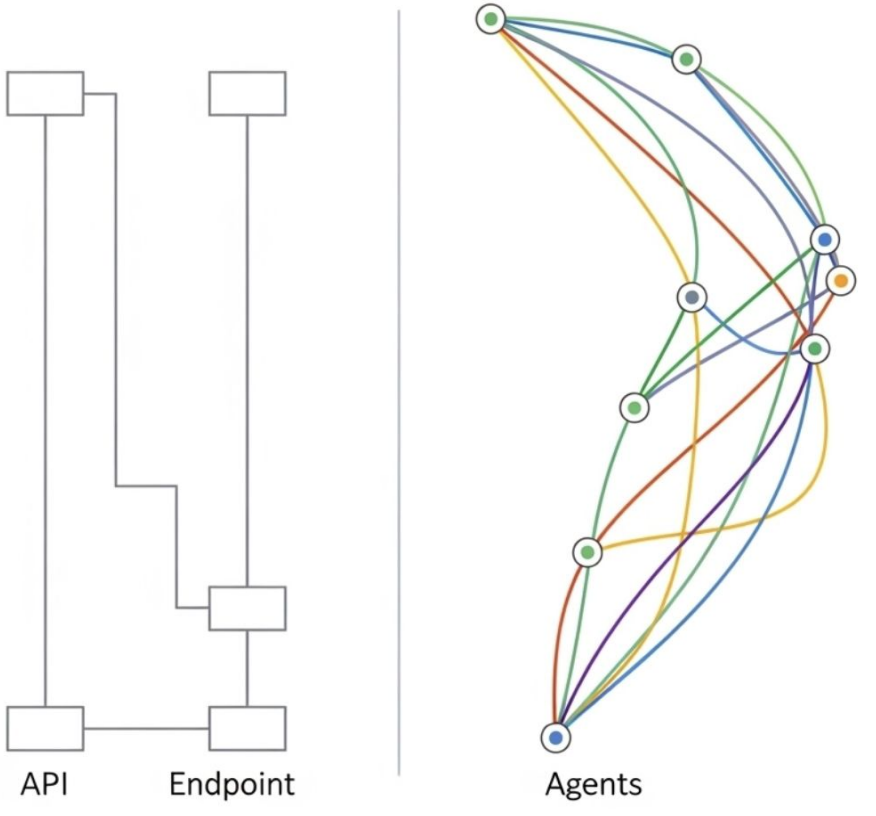
- AI agents are programs with autonomy
- They perceive environments and take actions
- Agents aim to achieve goals and adapt
- They utilize LLMs for advanced reasoning



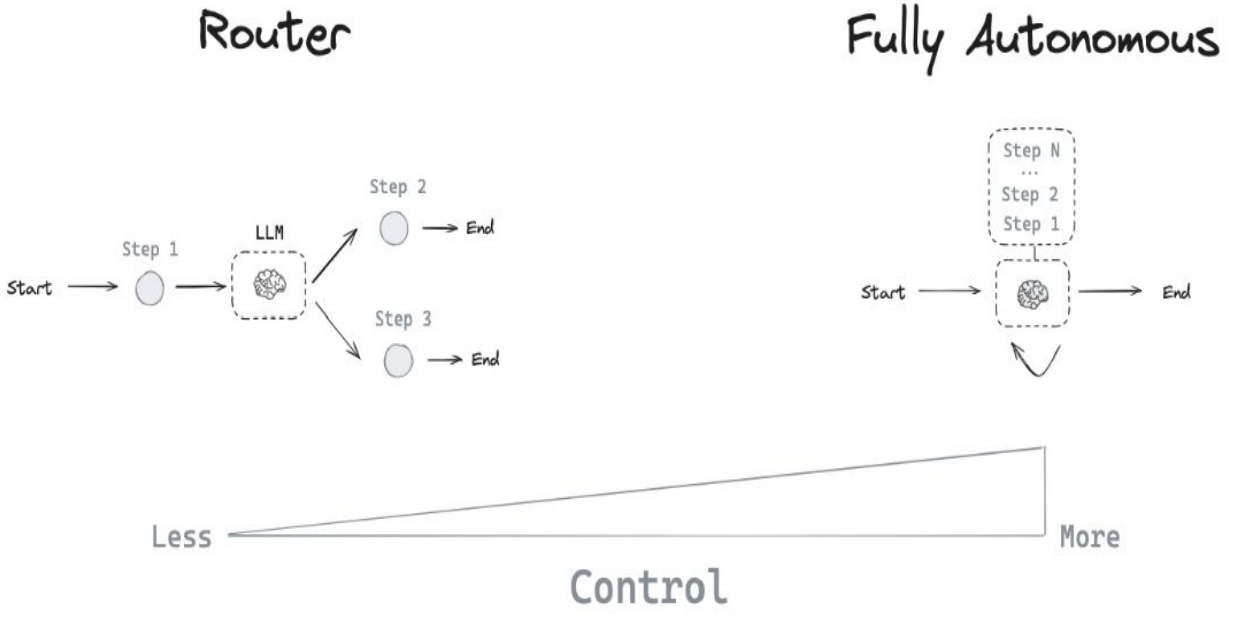
Components of an Agent



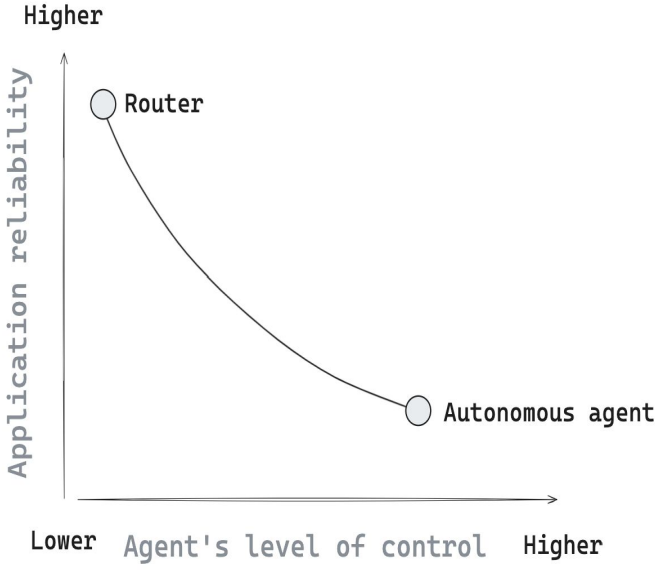
The agentic shift



Agentic Control



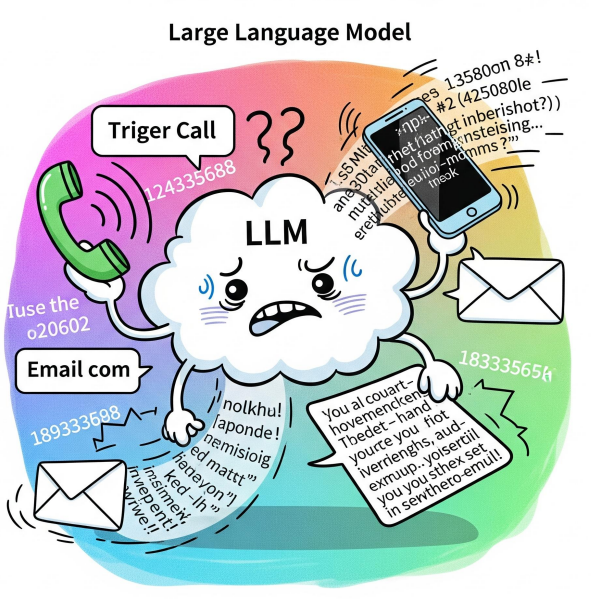
What this means for application reliability ?



The 12 Factor Framework

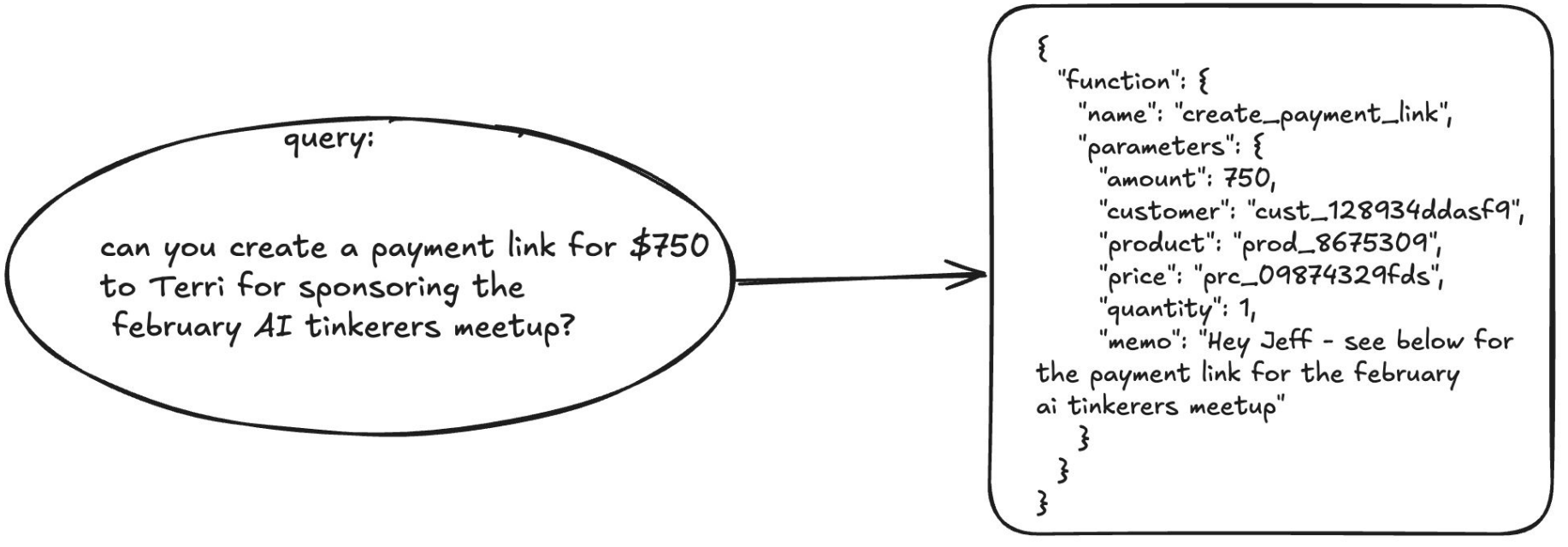
Unreliable Behaviour 1

1. Intent Misinterpretation and User Experience Degradation
2. The more the tools, the more the confusion, the more the tool hallucination
    a. I need to reach out to John about the quarterly report
        i. LLM Triggers email and phone call
3. Partial Tool input generation instead of all the data needed for tool execution
4. Scale issues [What to do if 1000 tools and how to debug ?]



And some yet to observe behaviours :)

#Factor 1: Natural Language to Tool Call



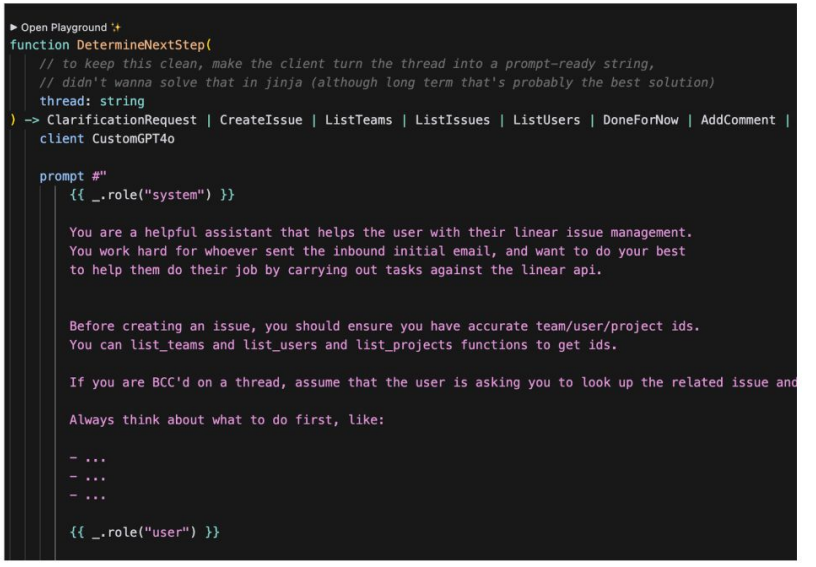
Unreliable Behaviour 2

1. Inconsistent Responses [It worked a minute back :( ]
2. Missing details due to Context Loss [It was present in the history]
3. Memory and Context Problems [What to remember and what not to ?]
4. Debugging nightmare :([So many messages to go through without structure]

And some yet to observe behaviours :)

Factor 2: Own your prompts (**)

```
Agent:
   role: str
   goal: str
   personality: str

Task:
  instructions: str
  expected_output: Type[BaseModel]
  tools: list[tool]


agent.run(task)
```



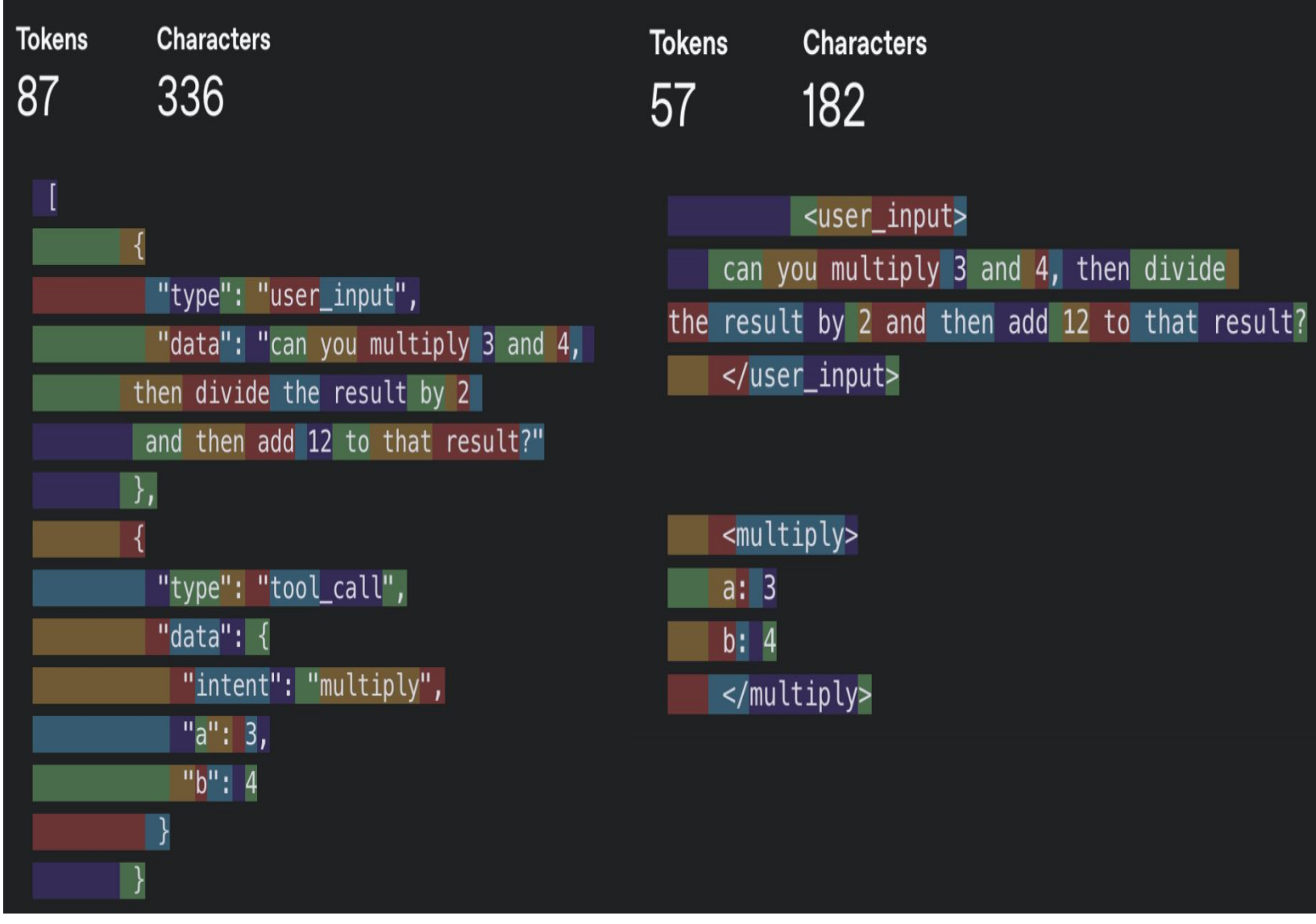
black box

full control

BOML

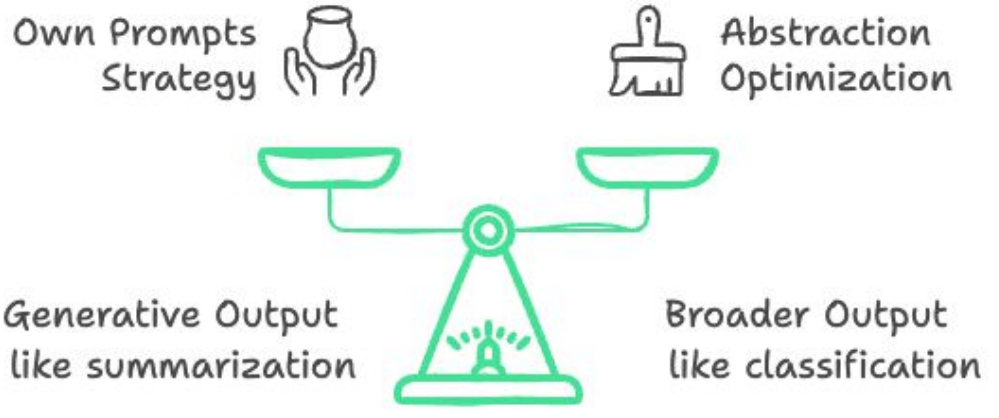
```
function DetermineNextStep(thread: string) -> DoneForNow | ListGitTags | DeployBackend | DeployFrontend | RequestMor
  prompt #"
    {{ _.role("system") }}

    You are a helpful assistant that manages deployments for frontend and backend systems.
    You work diligently to ensure safe and successful deployments by following best practices
    and proper deployment procedures.

    Before deploying any system, you should check:
    - The deployment environment (staging vs production)
    - The correct tag/version to deploy
    - The current system status

    You can use tools like deploy_backend, deploy_frontend, and check_deployment_status
    to manage deployments. For sensitive deployments, use request_approval to get
    human verification.

    Always think about what to do first, like:
    - Check current deployment status
    - Verify the deployment tag exists
    - Request approval if needed
    - Deploy to staging before production
    - Monitor deployment progress

    {{ _.role("user") }}

    {{ thread }}

    What should the next step be?
  "#
}
```

Tokens can be saved as well

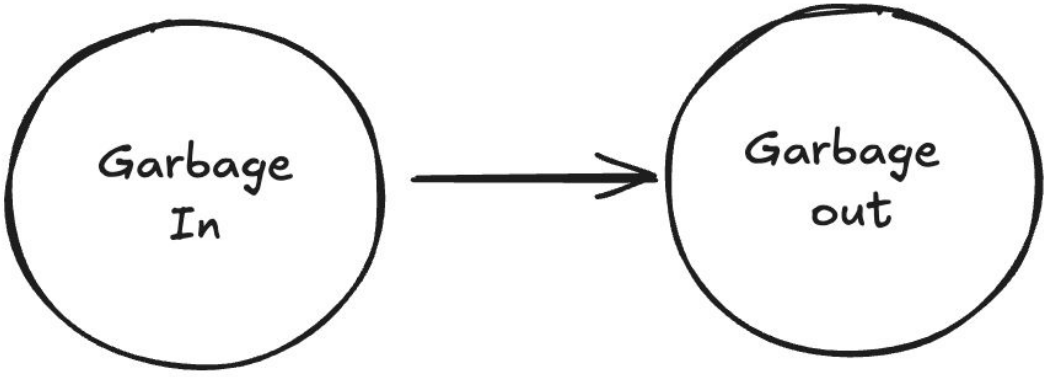


- AI prompts are like Jenga blocks; each one impacts the system's stability.
- Treat prompts as owned code, not throwaway text, for stable AI agent development.
- Version control and testing are crucial when modifying prompts to prevent system collapse.

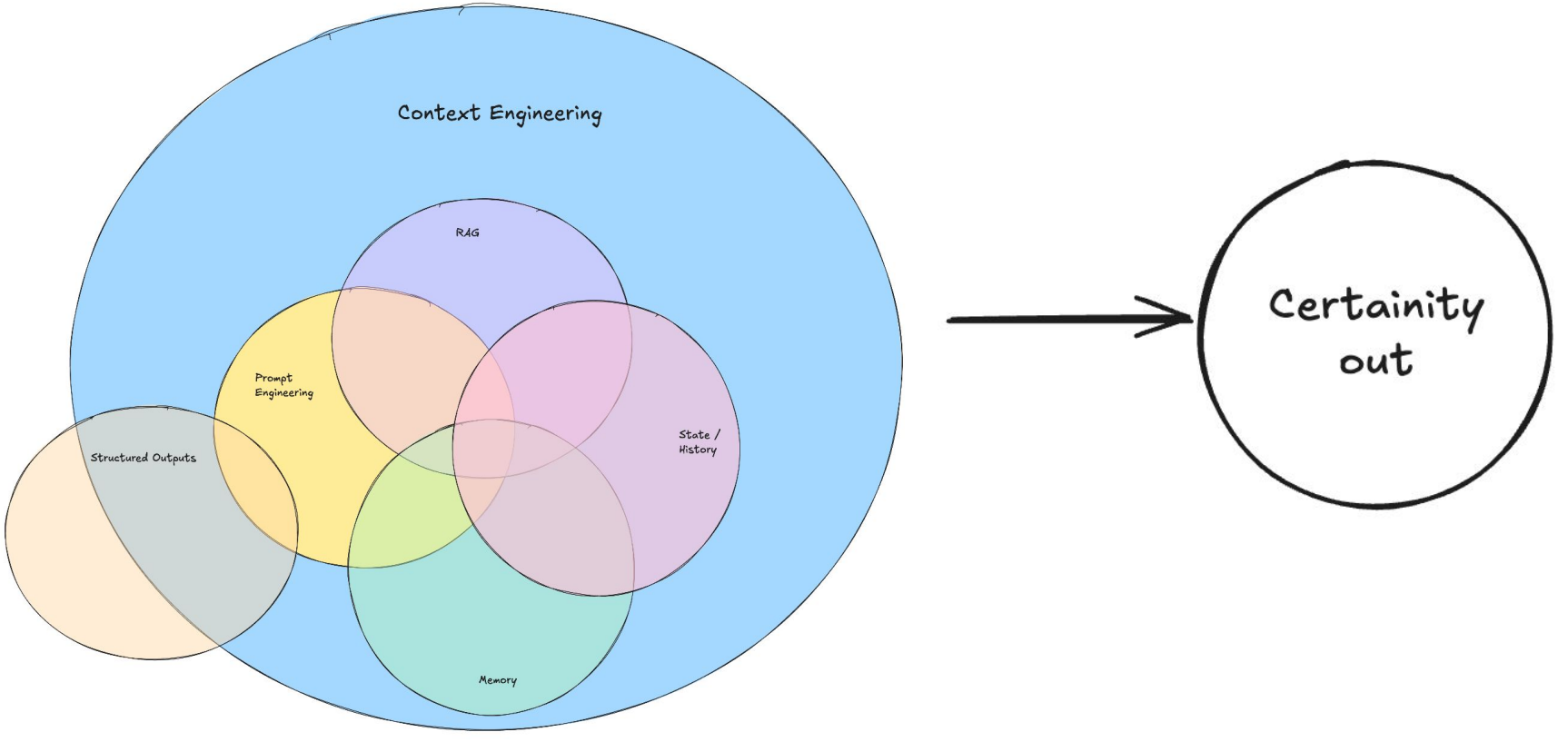
My opinion on factor 2



Unreliable Behaviour 3



Factor 3: Own your context window



Factor 4: Tools are just structured outputs

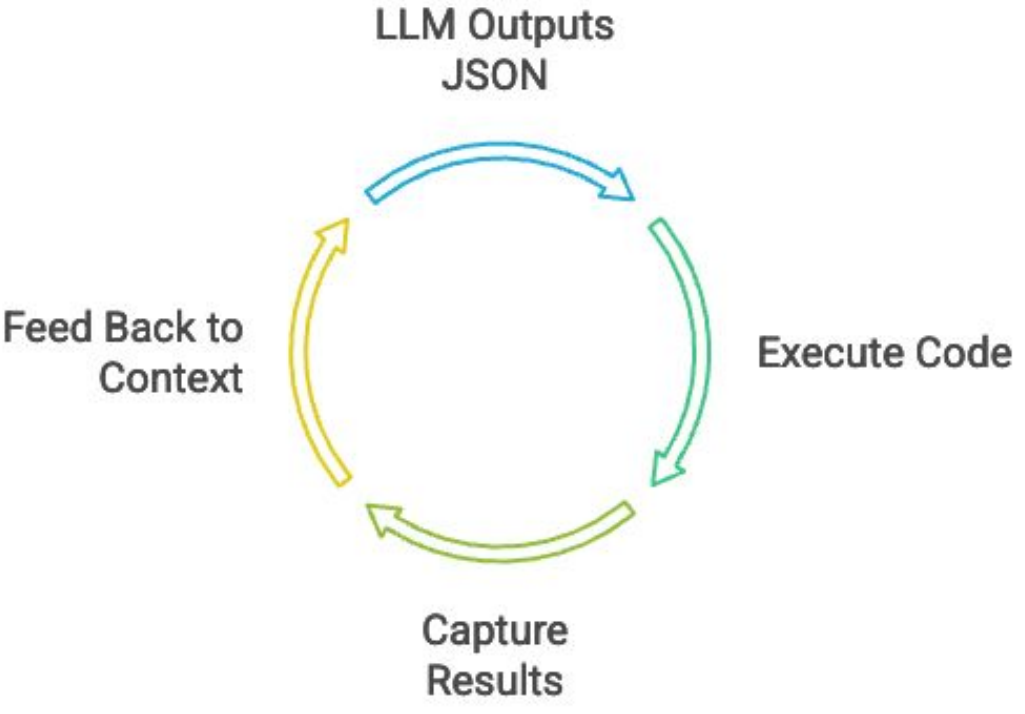
```
const openai_tools: ChatCompletionTool[] = [
  {
    type: "function",
    function: {
      name: "multiply",
      description: "multiply two numbers",
      parameters: {
        type: "object",
        properties: {
          a: { type: "number" },
          b: { type: "number" },
        },
        required: ["a", "b"],
      },
    },
  },
  {
    type: "function",
    function: {
      name: "add",
      description: "add two numbers",
      parameters: {
        type: "object",
        properties: {
          a: { type: "number" },
          b: { type: "number" },
        },
        required: ["a", "b"],
      },
    },
  },
];
```

JSON Schema

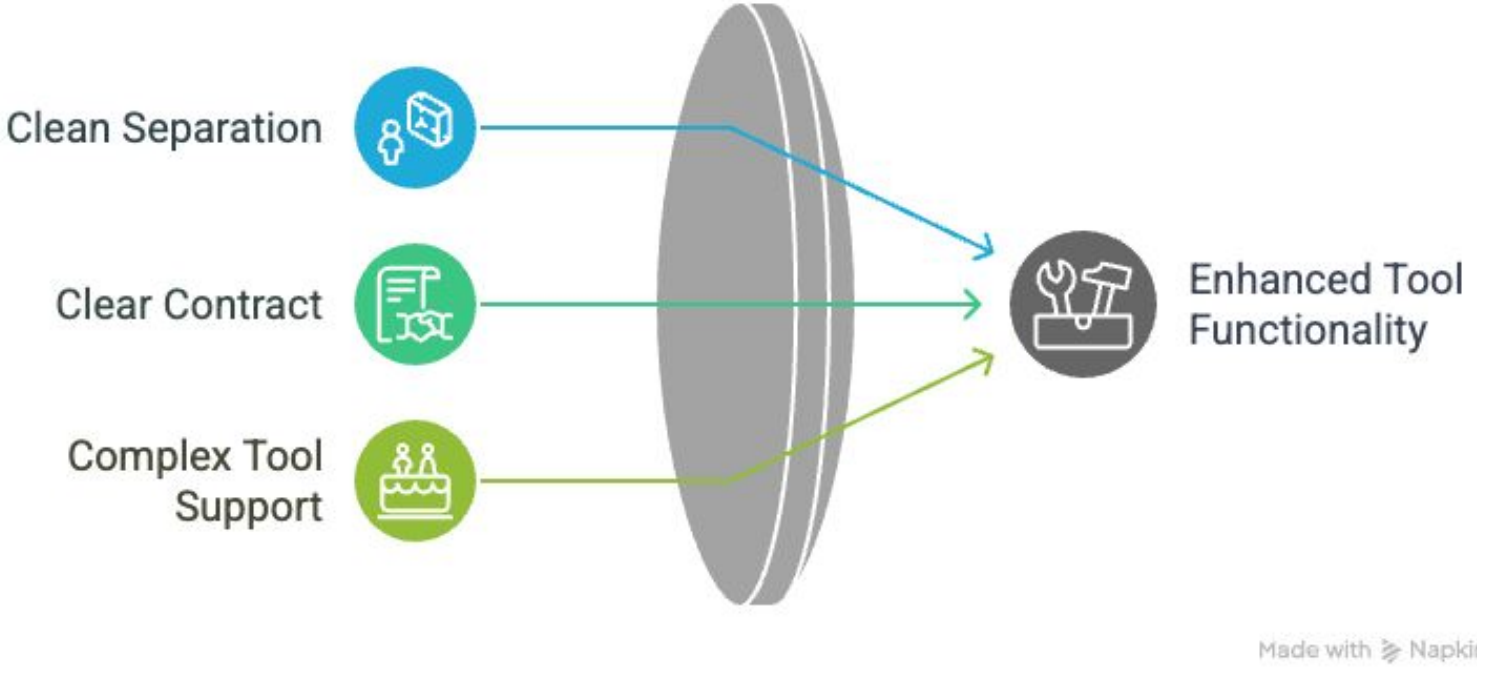
Prompt-Only

```
What should the next step be?

Answer in JSON using any of these schemas:
{
  // you can request more information from me
  intent: "request_more_information",
  message: string,
} or {
  intent: "create_issue",
  issue: {
    title: string,
    description: string,
    team_id: string,
    // name of the team to create the issue in
    team_name: string or null,
    project_id: string or null,
    // name of the project to create the issue in
    project_name: string or null,
    assignee_id: string or null,
    // name of the user to assign the issue to
    assignee_name: string or null,
    labels_ids: string[],
    labels_names: string[],
    // The priority of the issue.
    // 0 = No priority, 1 = Urgent, 2 = High, 3 = Normal, 4 = Low.
    priority: int or null,
  },
} or {
  intent: "list_teams",
  // what is your goal or desired outcome with this operation
  query_intent: string or null,
} or {
```

Code Execution Example



Benefits


Clean Separation
Clear Contract
Complex Tool Support
Enhanced Tool Functionality
Made with Napki

Unreliable Behaviour

1. Execution-Business State Misalignment
    a. For example, the execution state might show "waiting for approval" while the business state shows "approval already received," causing the agent to wait indefinitely for something that already occurred
2. Redundant Execution of flow

Factor 5: Unified Agent State

current step, next step, waiting status, retry counts, etc.

list of OpenAI messages, list of tool calls and results

Two States

```
class ExecutionState:
   current_step: int
   status: str  # "waiting", "executing", "paused"
   retry_count: int
   next_action: str

class BusinessState:
   messages: List[Message]
   tool_results: List[ToolResult]
   user_context: Dict
   completed_tasks: List[Task]
```

Single State

```python
@dataclass
class UnifiedStateEvent:
    timestamp: datetime
    role: str  # "user", "assistant", "tool", "system"
    content: str
    metadata: Optional[Dict[str, Any]] = None
    tool_calls: Optional[List[Dict]] = None
```

But…Seperate if..

Sensitive credentials

Credentials that should not be in context windows.

Long-running agents

Context windows become unwieldy for long-running agents.

Large data

Large data consumes too many tokens in states.

#Factor 6:

Agent APIs

1	2	3	4	5
Launch Agent	**Query Agent**	**Pause Agent**	**Resume Agent**	**Stop Agent**
Initiate the agent with a simple API	Check the agent's status or data	Halt the agent for long operations	Restart the agent from where it stopped	Terminate the agent's execution

Factor 7: Contact humans with tool calls

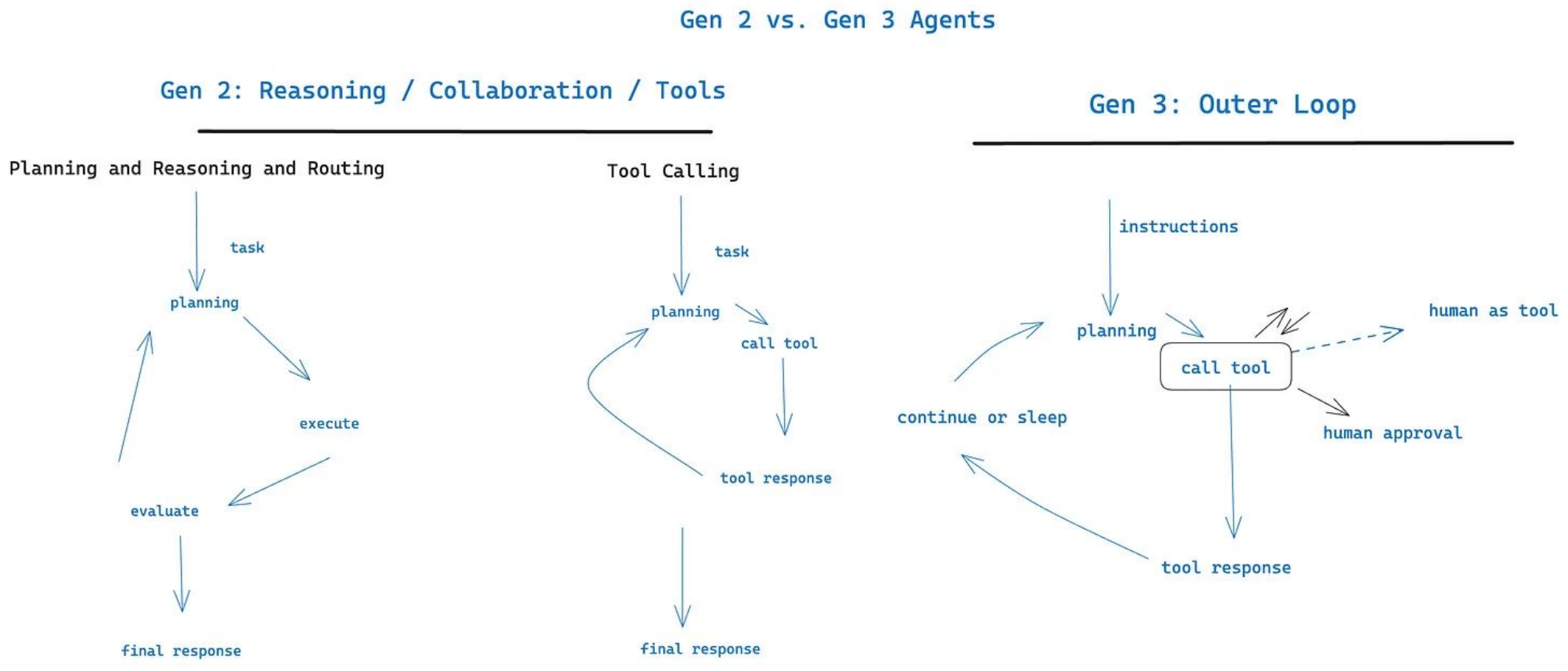
```
content: "the weather in tokyo is 57 and rainy"

OR

content: "for what postal code?"

OR

tool_calls: [...]
```

model switches between tools and plaintext

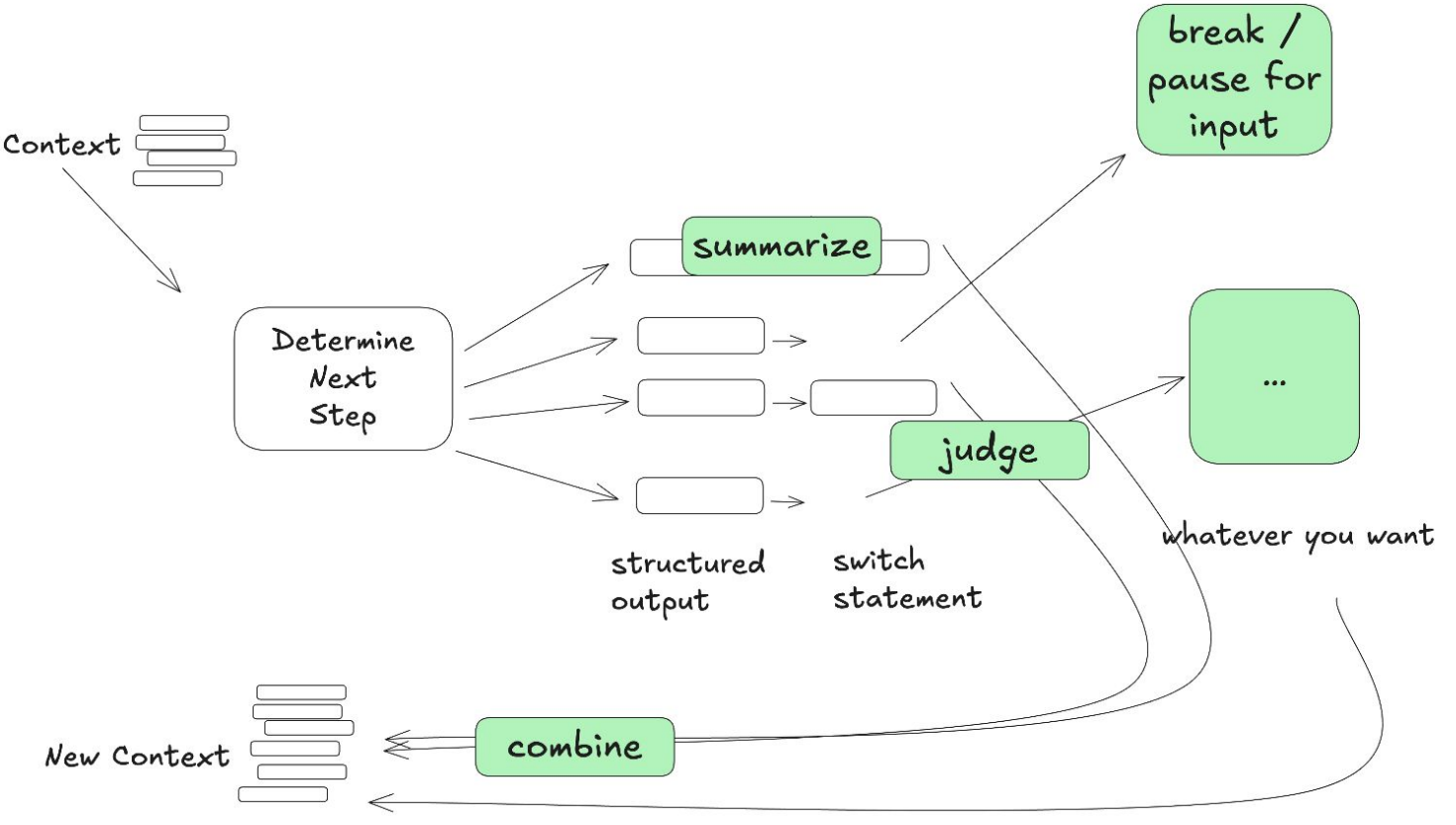
```
tool_calls: [{
  function: "final_answer",
  arguments: {
    message: "the weather in tokyo is 57 and rainy",
  }
}]

OR

tool_calls: [{
  function: "request_clarification",
  arguments: {
    message: "for what postal code?",
  }
}]

OR

tool_calls: {
  function: "check_weather_in_city",
  arguments: {
    "city": "tokyo",
    "postal_code": "12345"
}
```

everything is tools

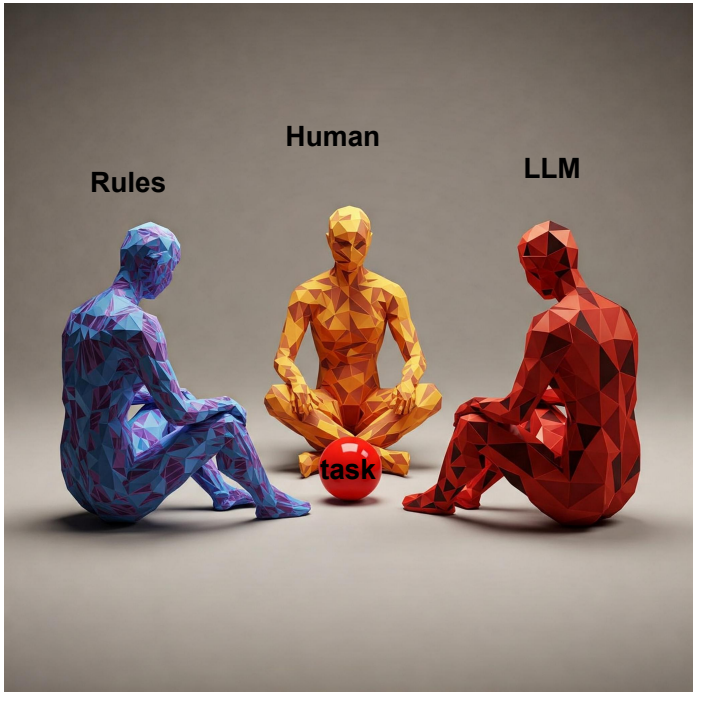
Reduced context switch overhead



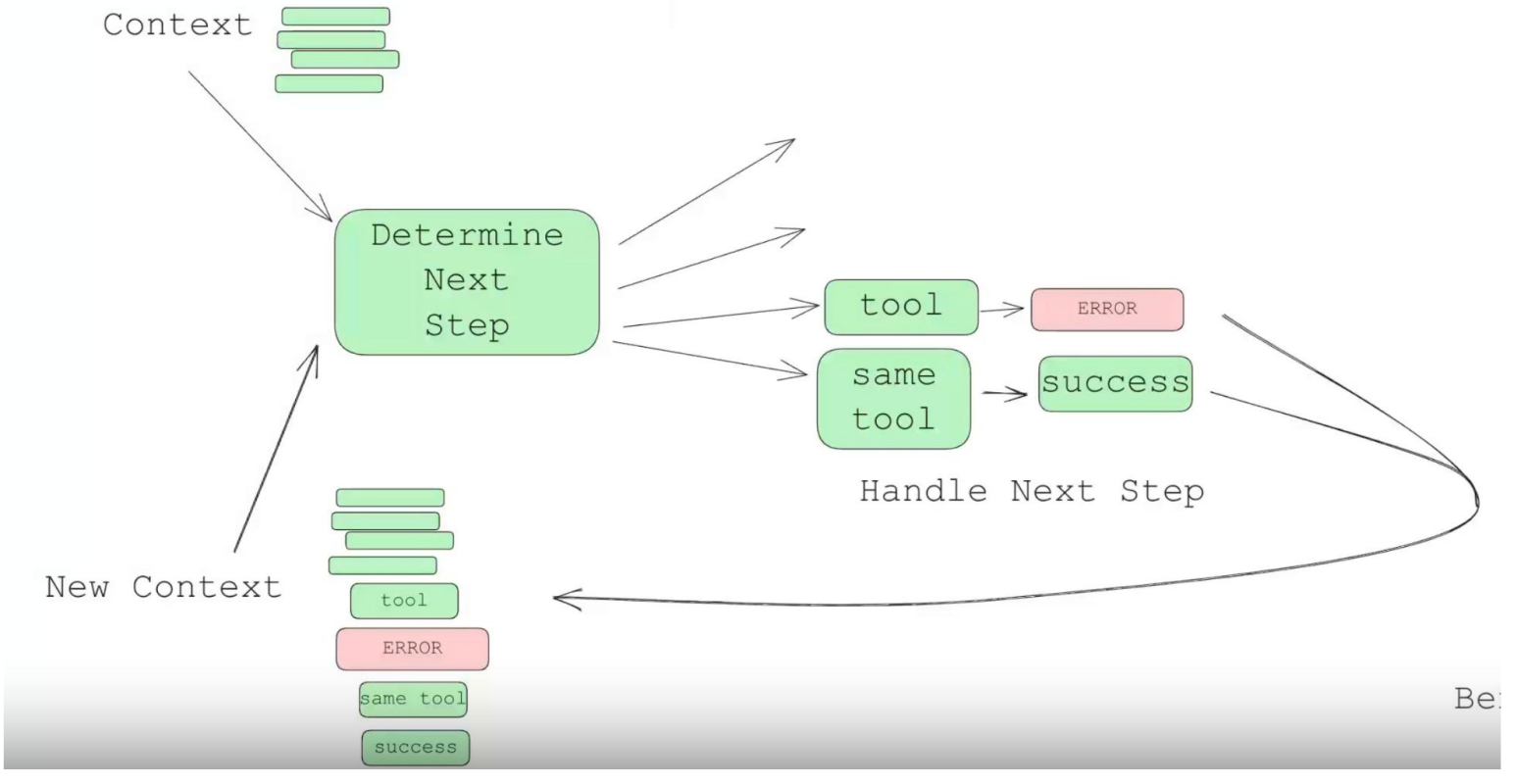
Factor 8: Own your control flow



The three actors analogy for building reliable agents!



Factor 9 : Compact Errors into Context Window

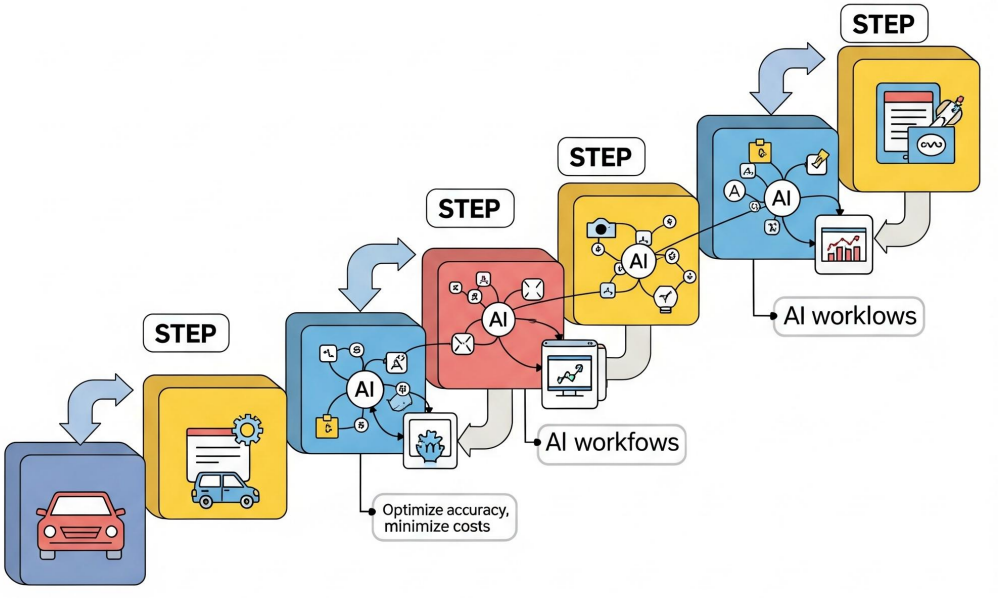


Factor 10: Small and Focused Agents

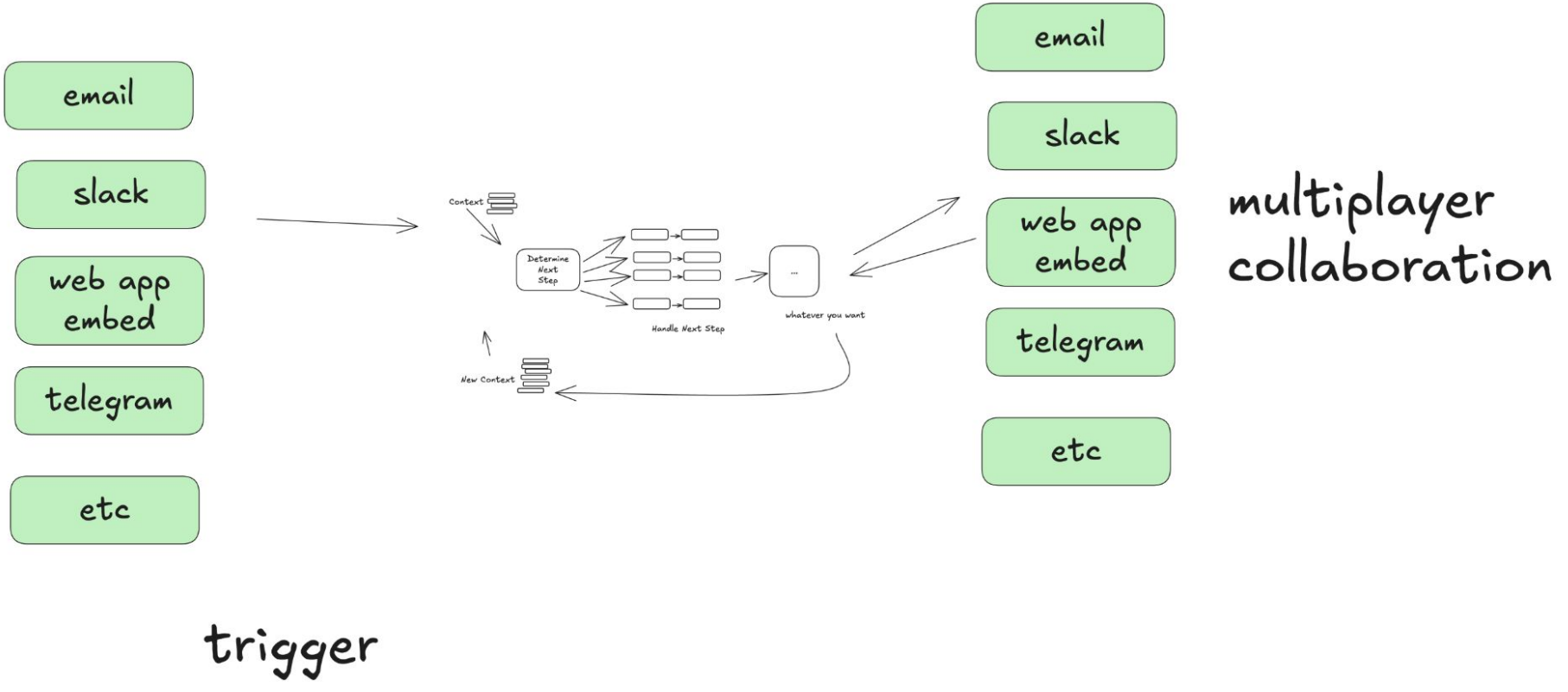
- Build small, focused agents for specific tasks.
- Avoid monolithic agents trying to do everything.
- Smaller agents keep context windows manageable.
- Limit agent tasks to 3-10, or maximum 20 steps.
- This improves LLM performance and reduces focus loss.

Flow engineering: AlphaCodium case

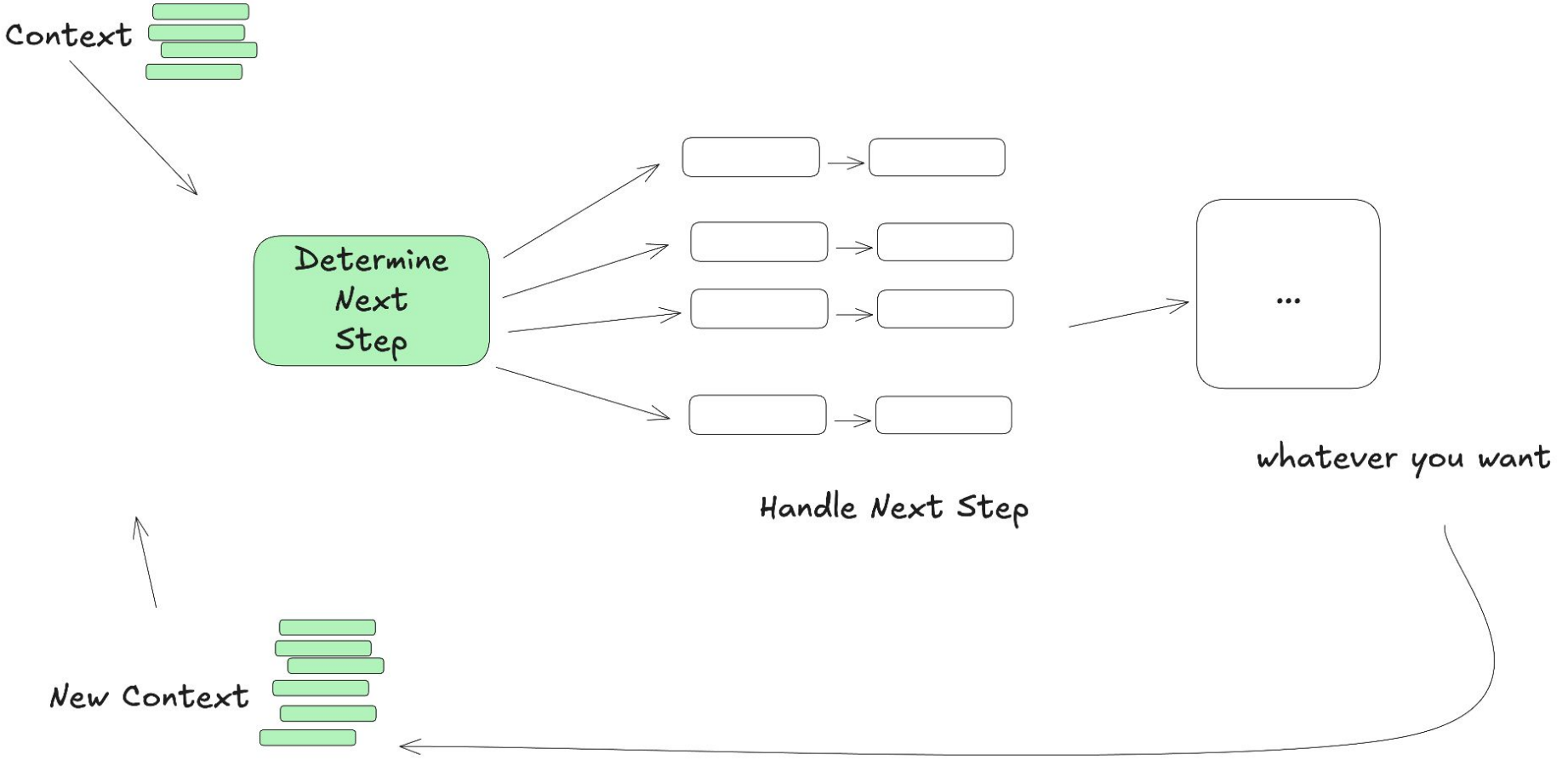
- Orchestrated AI workflows enable focused agents.
- Complex tasks break into manageable sequential steps.
- Improves accuracy and optimizes cost significantly.
- AlphaCodium leveraged this for superior performance.



Factor 11: Trigger from anywhere, meet users where they are



Factor 12 : Make your agent a stateless reducer

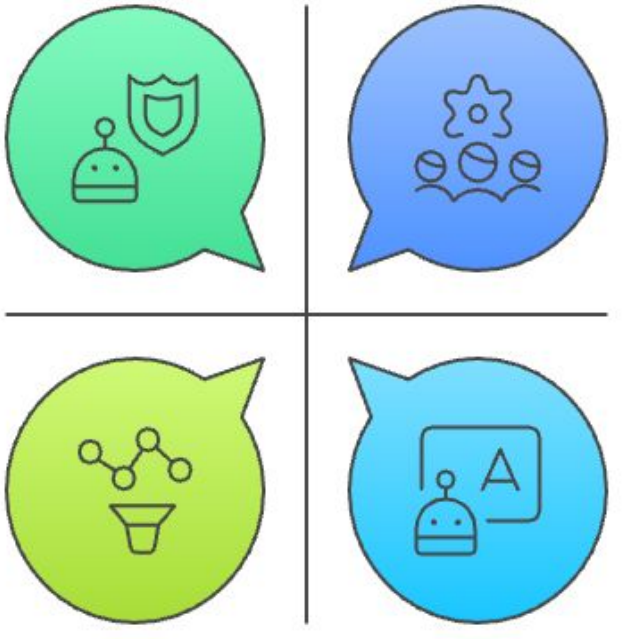


The extra factors

1

Safety, Guardrails & Human-in-the-Loop

Safety, Guardrails & Human-in-the-Loop provide robust security at a cost.



2

Planner/Verifier Pattern

Planner/Verifier Pattern ensures quality through complex validation processes.

3

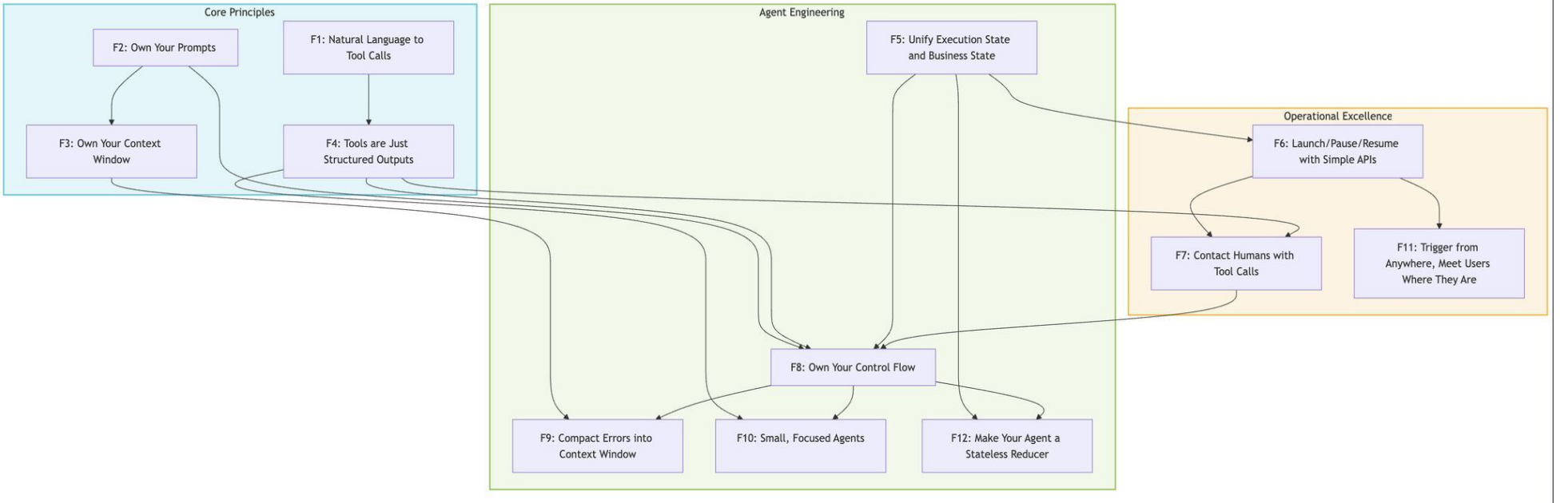
Leverage Foundation Model Capabilities

Leverage Foundation Model Capabilities maximizes efficiency with minimal cost.

4

Continuous Evaluation

Continuous Evaluation addresses non-deterministic behavior with moderate complexity.


Core Principles
F2: Own Your Prompts
F1: Natural Language to Tool Calls
F3: Own Your Context Window
F4: Tools are Just Structured Outputs
Agent Engineering
F5: Unify Execution State and Business State
F8: Own Your Control Flow
F9: Compact Errors into Context Window
F10: Small, Focused Agents
F12: Make Your Agent a Stateless Reducer
Operational Excellence
F6: Launch/Pause/Resume with Simple APIs
F7: Contact Humans with Tool Calls
F11: Trigger from Anywhere, Meet Users Where They Are

https://www.linkedin.com/in/poornagurram/

aiengineerinsights.com