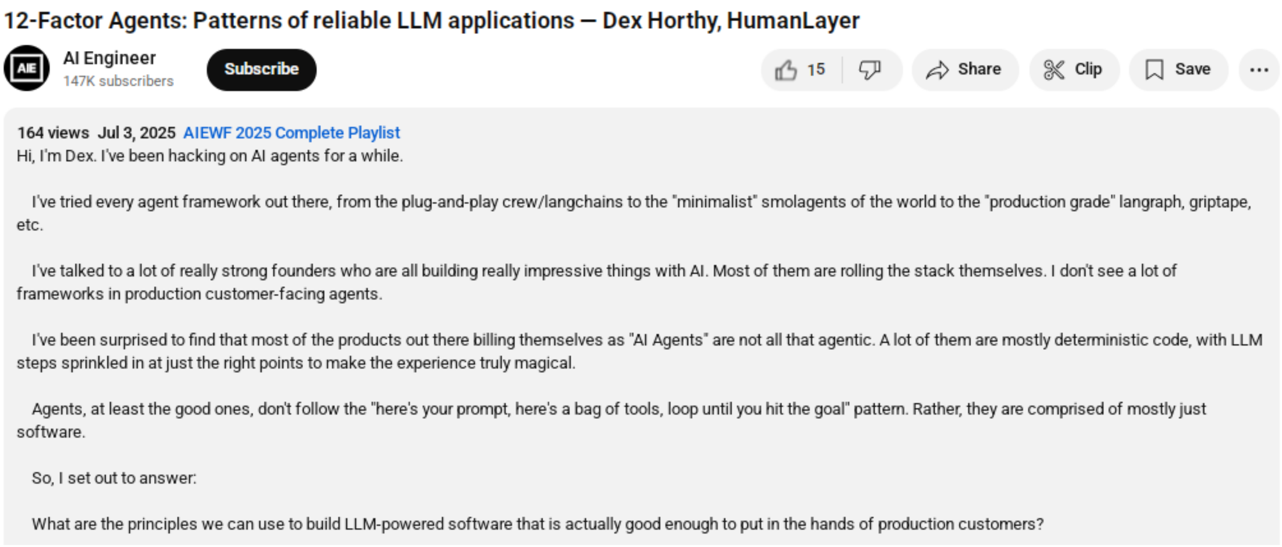


12-Factor Agents: Patterns of reliable LLM applications — Dex Horthy, HumanLayer

164 views Jul 3, 2025 AIEWF 2025 Complete Playlist

Hi, I'm Dex. I've been hacking on AI agents for a while.

I've tried every agent framework out there, from the plug-and-play crew/langchains to the "minimalist" smolagents of the world to the "production grade" langraph, griptape, etc.

I've talked to a lot of really strong founders who are all building really impressive things with AI. Most of them are rolling the stack themselves. I don't see a lot of frameworks in production customer-facing agents.

I've been surprised to find that most of the products out there billing themselves as "AI Agents" are not all that agentic. A lot of them are mostly deterministic code, with LLM steps sprinkled in at just the right points to make the experience truly magical.

Agents, at least the good ones, don't follow the "here's your prompt, here's a bag of tools, loop until you hit the goal" pattern. Rather, they are comprised of mostly just software.

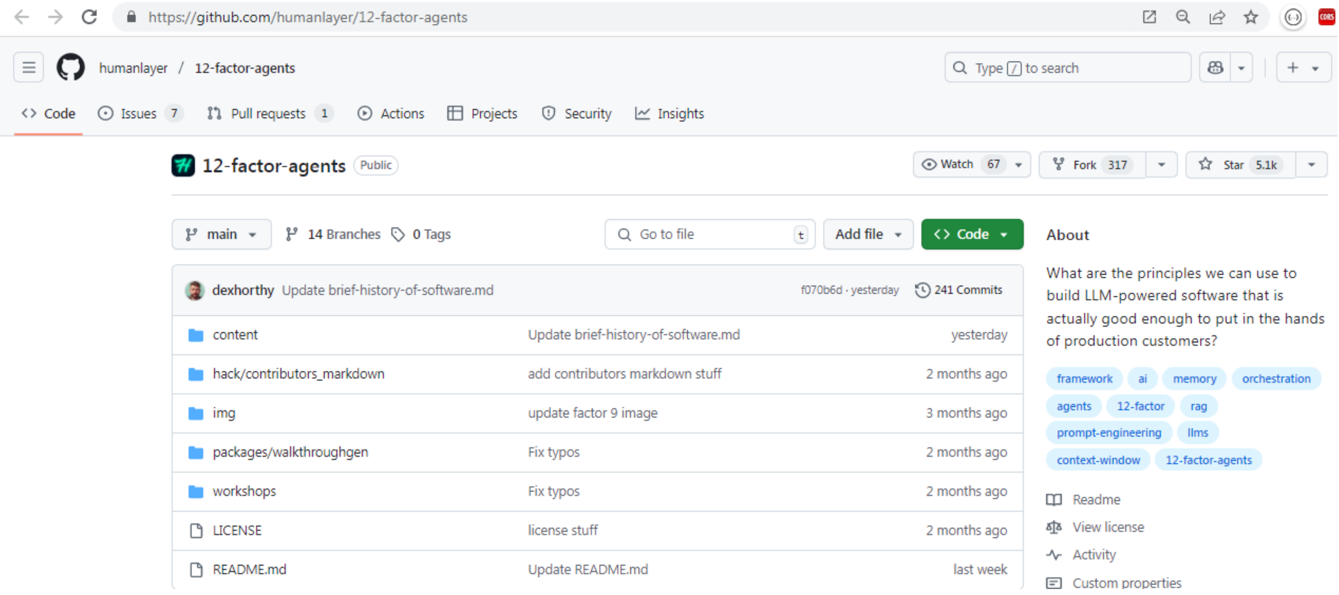
So, I set out to answer:

What are the principles we can use to build LLM-powered software that is actually good enough to put in the hands of production customers?

The Short Version: The 12 Factors

Even if LLMs continue to get exponentially more powerful, there will be core engineering techniques that make LLM-powered software more reliable, more scalable, and easier to maintain.

How We Got Here: A Brief History of Software
Factor 1: Natural Language to Tool Calls
Factor 2: Own your prompts
Factor 3: Own your context window
Factor 4: Tools are just structured outputs
Factor 5: Unify execution state and business state
Factor 6: Launch/Pause/Resume with simple APIs
Factor 7: Contact humans with tool calls
Factor 8: Own your control flow
Factor 9: Compact Errors into Context Window
Factor 10: Small, Focused Agents
Factor 11: Trigger from anywhere, meet users where they are
Factor 12: Make your agent a stateless reducer

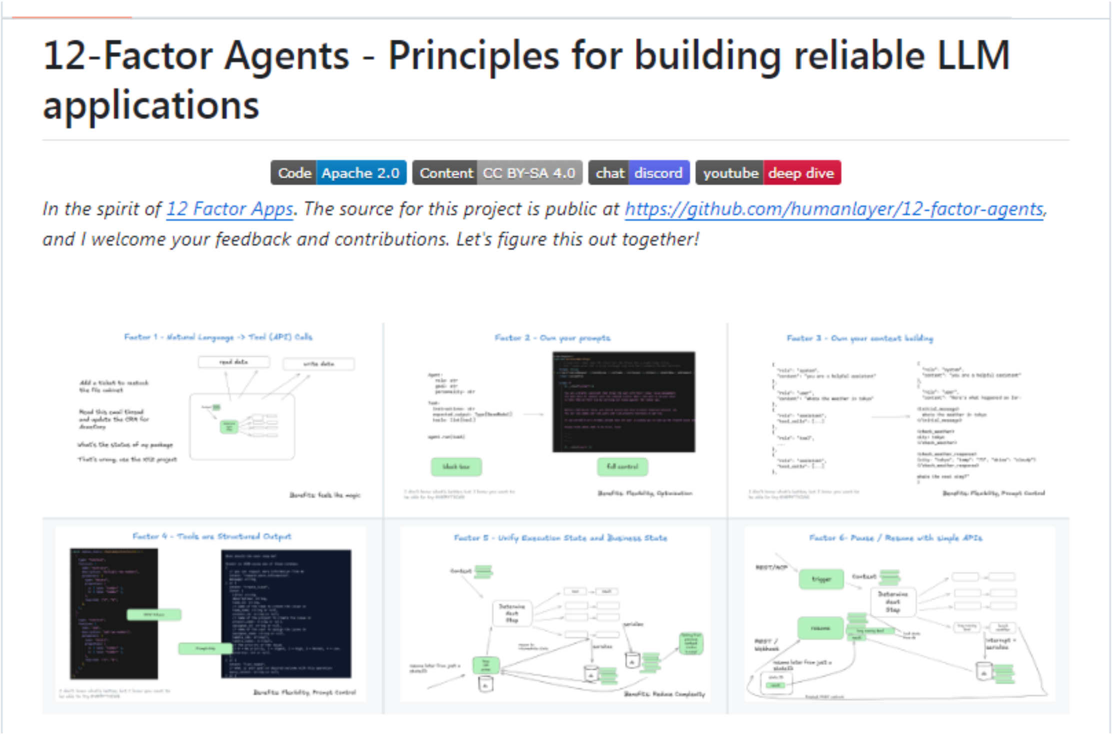


dexhorthy Update brief-history-of-software.md f070b6d · yesterday 241 Commits

content	Update brief-history-of-software.md	yesterday
hack/contributors_markdown	add contributors markdown stuff	2 months ago
img	update factor 9 image	3 months ago
packages/walkthroughgen	Fix typos	2 months ago
workshops	Fix typos	2 months ago
LICENSE	license stuff	2 months ago
README.md	Update README.md	last week

What are the principles we can use to build LLM-powered software that is actually good enough to put in the hands of production customers?

framework ai memory orchestration agents 12-factor rag prompt-engineering llms context-window 12-factor-agents

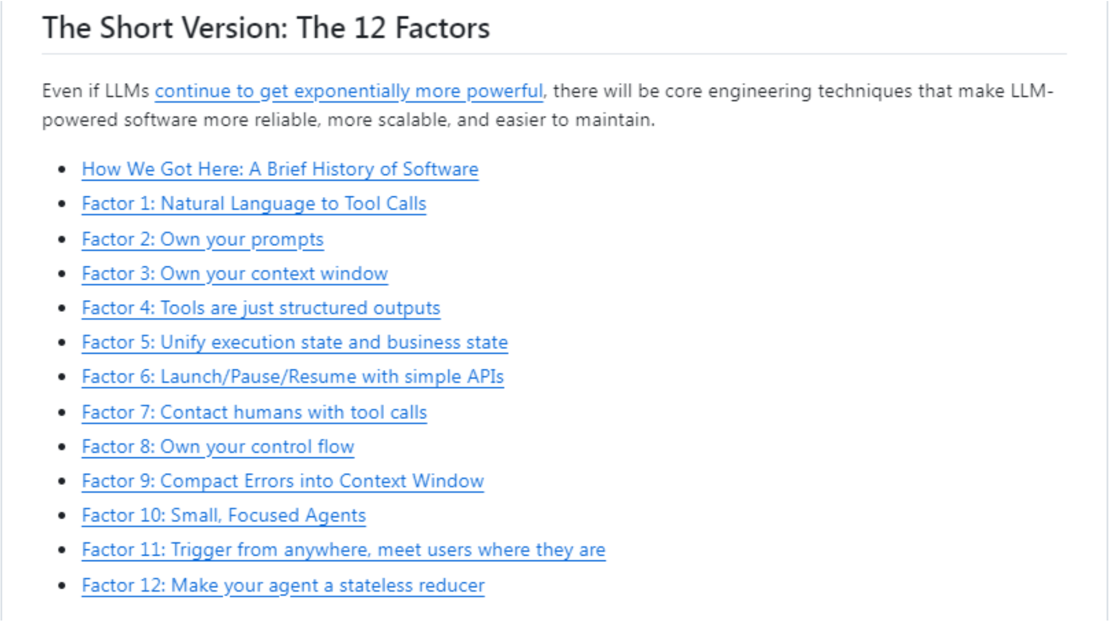


12-Factor Agents - Principles for building reliable LLM applications

Code Apache 2.0 | Content CC BY-SA 4.0 | chat discord | youtube deep dive

In the spirit of 12 Factor Apps. The source for this project is public at https://github.com/humanlayer/12-factor-agents, and I welcome your feedback and contributions. Let's figure this out together!

Factor 1 - Natural Language -> Tool (API) Calls
read data
write data
Add a ticket to restock the file cabinet
Read this email thread and update the CRM for Acme Corp
what's the status of my package
That's wrong, use the XYZ project
Benefits: feels like magic
Factor 2 - Own your prompts
black box
full control
Benefits: Flexibility, Optimization
Factor 3 - Own your context building
Benefits: Flexibility, Prompt Control
Factor 4 - Tools are Structured Output
Benefits: Flexibility, Prompt Control
Factor 5 - Unify Execution State and Business State
Context
Determine Next Step
serialize
Benefits: Reduce Complexity
Factor 6- Pause / Resume with simple APIs
REST/ACP
trigger
Context
Determine Next Step
resume
REST / Webhook
interrupt + serialize



The Short Version: The 12 Factors

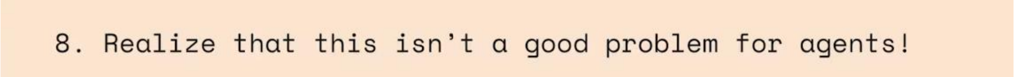
Even if LLMs continue to get exponentially more powerful, there will be core engineering techniques that make LLM-powered software more reliable, more scalable, and easier to maintain.

- How We Got Here: A Brief History of Software
- Factor 1: Natural Language to Tool Calls
- Factor 2: Own your prompts
- Factor 3: Own your context window
- Factor 4: Tools are just structured outputs
- Factor 5: Unify execution state and business state
- Factor 6: Launch/Pause/Resume with simple APIs
- Factor 7: Contact humans with tool calls
- Factor 8: Own your control flow
- Factor 9: Compact Errors into Context Window
- Factor 10: Small, Focused Agents
- Factor 11: Trigger from anywhere, meet users where they are
- Factor 12: Make your agent a stateless reducer

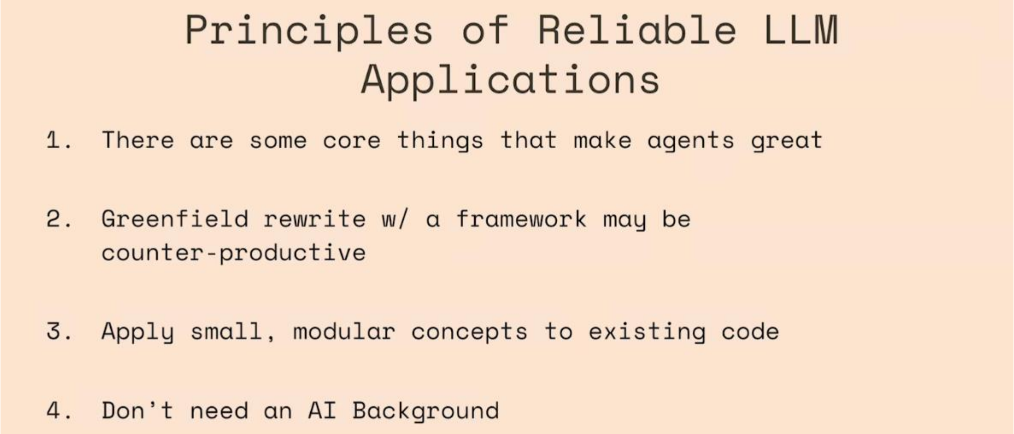
Building Agents in 2025

The Agent Journey

1. Decide you want to build an agent
2. Product design, UX mapping, what problems to solve
3. Want to move fast, so grab $FRAMEWORK and *get to building*
4. Get to 70-80% quality bar
5. Realize that 80% isn't good enough
6. Realize that getting past 80% requires reverse-engineering the framework, prompts, flow, etc.
7. Start over from scratch



8. Realize that this isn’t a good problem for agents!



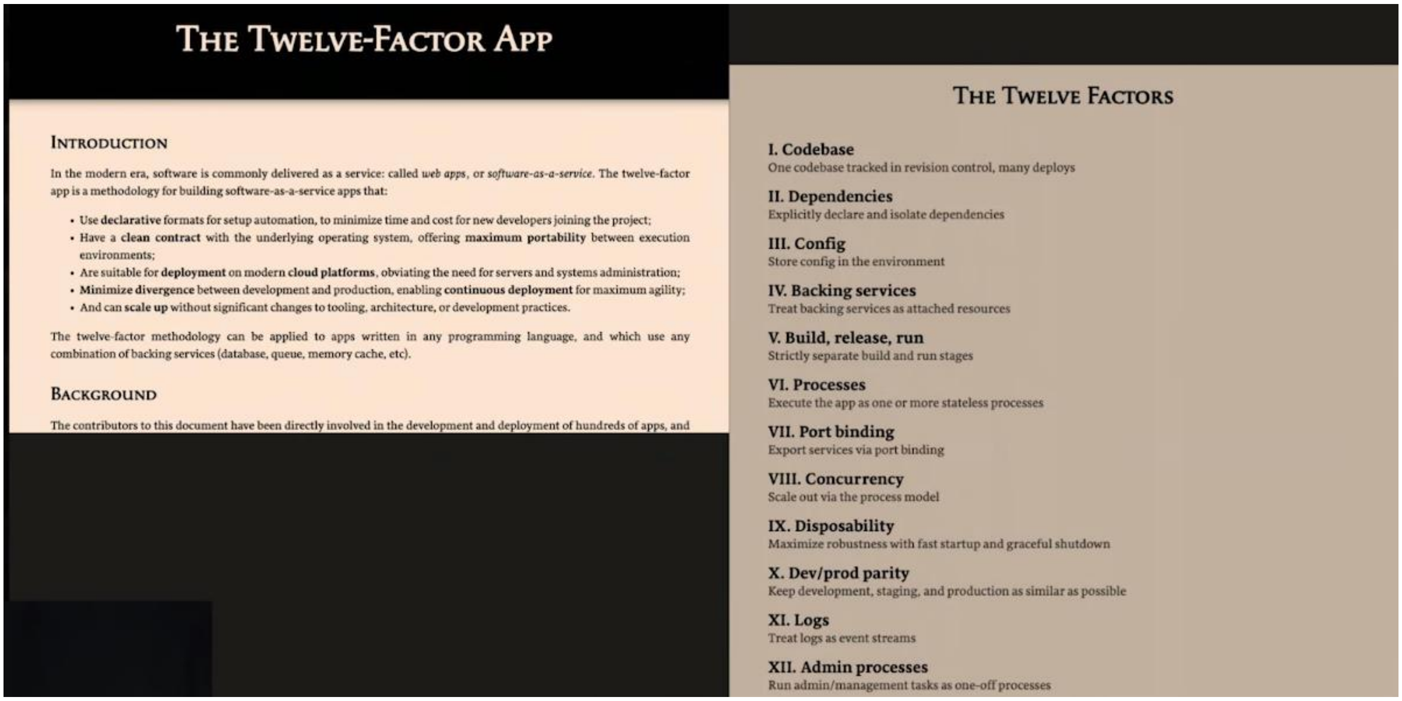
Principles of Reliable LLM Applications

1. There are some core things that make agents great

2. Greenfield rewrite w/ a framework may be counter-productive

3. Apply small, modular concepts to existing code

4. Don’t need an AI Background



The Twelve-Factor App

Introduction

In the modern era, software is commonly delivered as a service: called *web apps*, or *software-as-a-service*. The twelve-factor app is a methodology for building software-as-a-service apps that:

- Use **declarative** formats for setup automation, to minimize time and cost for new developers joining the project;
- Have a **clean contract** with the underlying operating system, offering **maximum portability** between execution environments;
- Are suitable for **deployment** on modern **cloud platforms**, obviating the need for servers and systems administration;
- **Minimize divergence** between development and production, enabling **continuous deployment** for maximum agility;
- And can **scale up** without significant changes to tooling, architecture, or development practices.

The twelve-factor methodology can be applied to apps written in any programming language, and which use any combination of backing services (database, queue, memory cache, etc).

Background

The contributors to this document have been directly involved in the development and deployment of hundreds of apps, and

The Twelve Factors

I. Codebase
One codebase tracked in revision control, many deploys

II. Dependencies
Explicitly declare and isolate dependencies

III. Config
Store config in the environment

IV. Backing services
Treat backing services as attached resources

V. Build, release, run
Strictly separate build and run stages

VI. Processes
Execute the app as one or more stateless processes

VII. Port binding
Export services via port binding

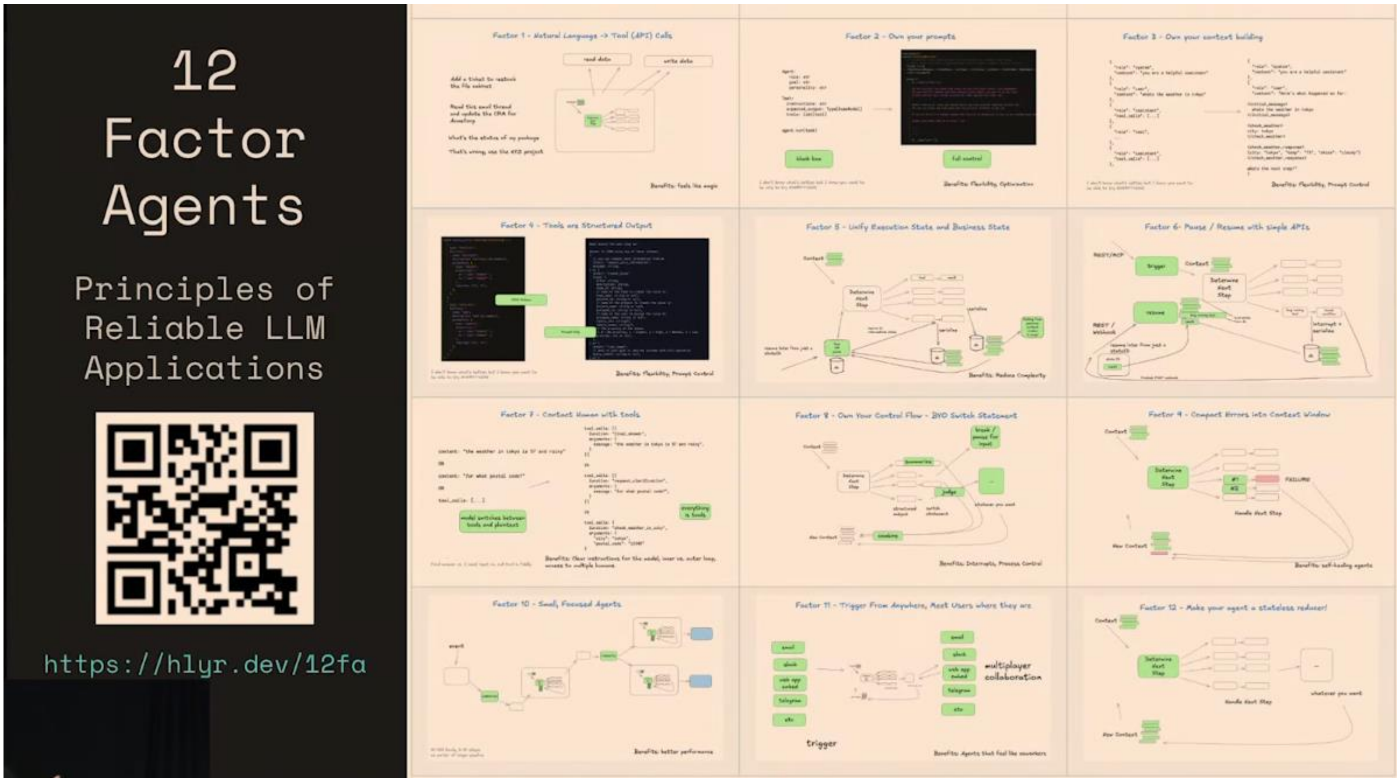
VIII. Concurrency
Scale out via the process model

IX. Disposability
Maximize robustness with fast startup and graceful shutdown

X. Dev/prod parity
Keep development, staging, and production as similar as possible

XI. Logs
Treat logs as event streams

XII. Admin processes
Run admin/management tasks as one-off processes



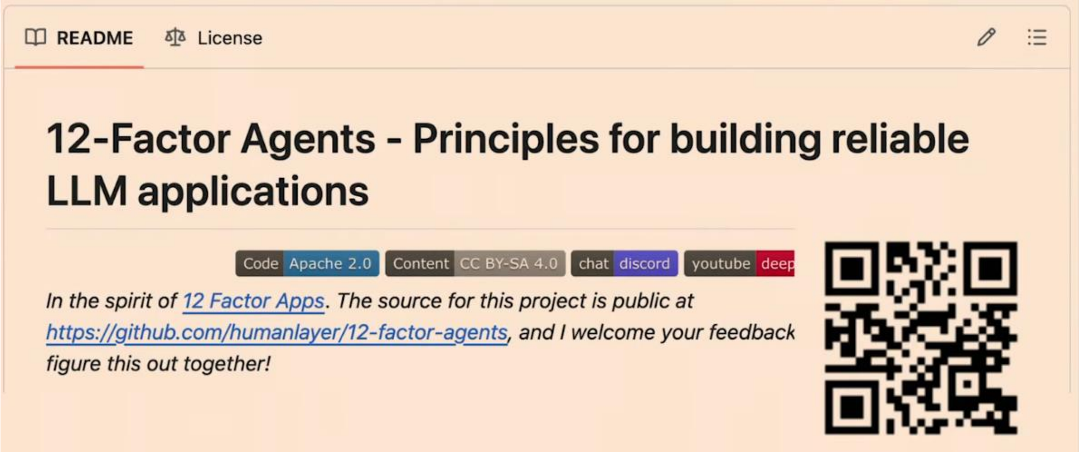
12 Factor Agents

Principles of Reliable LLM Applications

https://hlyr.dev/12fa

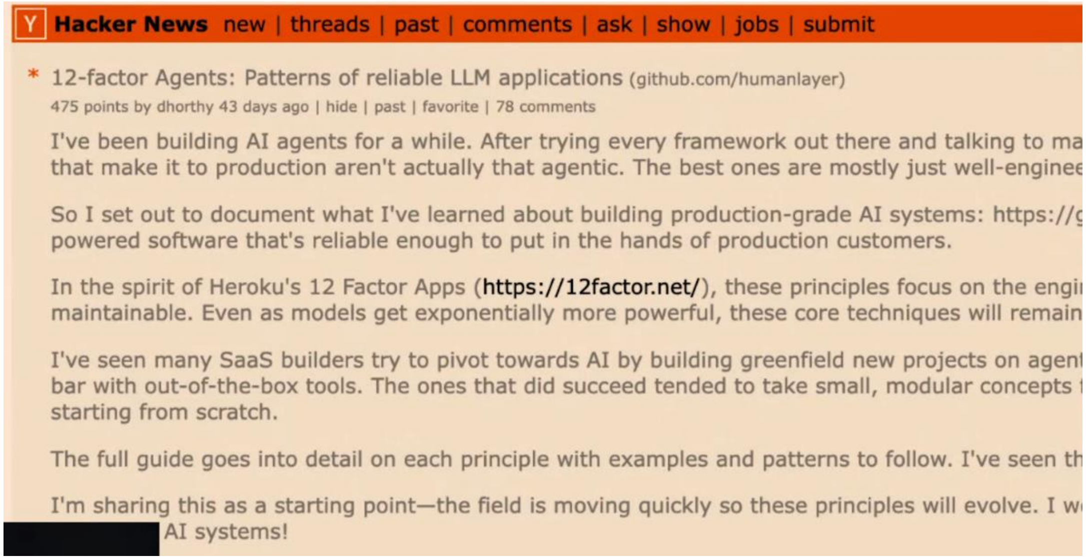
[redacted]

Factor 1 - Natural Language -> Tool (API) Calls
Benefits: tools like magic
Factor 2 - Own your prompts
black box
full control
Benefits: Flexibility, Optimization
Factor 3 - Own your context building
Benefits: Flexibility, Prompt Control
Factor 4 - Tools are Structured Output
Benefits: Flexibility, Prompt Control
Factor 5 - Unify Execution State and Business State
Benefits: Reduce Complexity
Factor 6- Pause / Resume with simple APIs
REST/MCP
trigger
Context
Determine Next Step
REST / Webhook
Factor 7 - Contact Human with tools
model switches between tools and plaintext
everything is tools
Benefits: Clear instructions for the model, inner vs. outer loop, access to multiple humans
Factor 8 - Own Your Control Flow - BYO Switch Statement
break / pause for input
judge
whatever you want
Benefits: Interrupts, Process Control
Factor 9 - Compact Errors into Context Window
Context
Determine Next Step
FAILURE
Handle Next Step
New Context
Benefits: self-healing agents
Factor 10 - Small, Focused Agents
Benefits: better performance
Factor 11 - Trigger From Anywhere, Meet Users where they are
email
slack
web app
telegram
etc
multiplayer collaboration
trigger
Benefits: Agents that feel like coworkers
Factor 12 - Make your agent a stateless reducer!
Context
Determine Next Step
Handle Next Step
whatever you want
New Context



12-Factor Agents - Principles for building reliable LLM applications

In the spirit of 12 Factor Apps. The source for this project is public at https://github.com/humanlayer/12-factor-agents, and I welcome your feedback figure this out together!



Y Hacker News new | threads | past | comments | ask | show | jobs | submit

* 12-factor Agents: Patterns of reliable LLM applications (github.com/humanlayer)

475 points by dhorthy 43 days ago | hide | past | favorite | 78 comments

I've been building AI agents for a while. After trying every framework out there and talking to ma
that make it to production aren't actually that agentic. The best ones are mostly just well-enginee

So I set out to document what I've learned about building production-grade AI systems: https://g
powered software that's reliable enough to put in the hands of production customers.

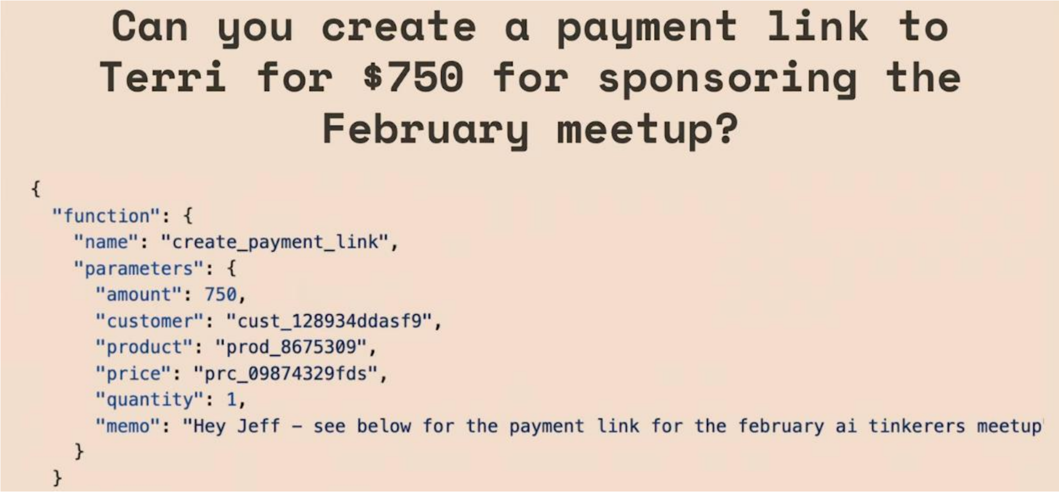
In the spirit of Heroku's 12 Factor Apps (https://12factor.net/), these principles focus on the engi
maintainable. Even as models get exponentially more powerful, these core techniques will remain

I've seen many SaaS builders try to pivot towards AI by building greenfield new projects on agent
bar with out-of-the-box tools. The ones that did succeed tended to take small, modular concepts
starting from scratch.

The full guide goes into detail on each principle with examples and patterns to follow. I've seen th

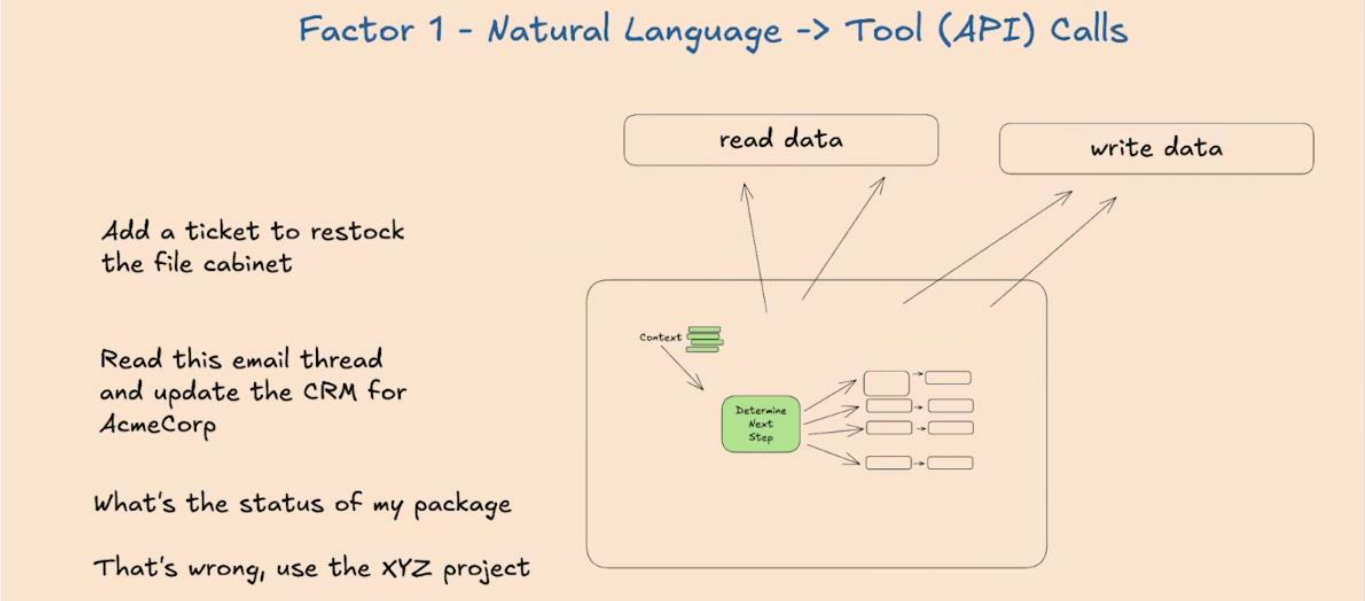
I'm sharing this as a starting point—the field is moving quickly so these principles will evolve. I w
[redacted] AI systems!

Factor 1
Natural Language to tool calls



Can you create a payment link to Terri for $750 for sponsoring the February meetup?

```
{
  "function": {
    "name": "create_payment_link",
    "parameters": {
      "amount": 750,
      "customer": "cust_128934ddasf9",
      "product": "prod_8675309",
      "price": "prc_09874329fds",
      "quantity": 1,
      "memo": "Hey Jeff - see below for the payment link for the february ai tinkerers meetup
    }
  }
```



Factor 1 - Natural Language -> Tool (API) Calls

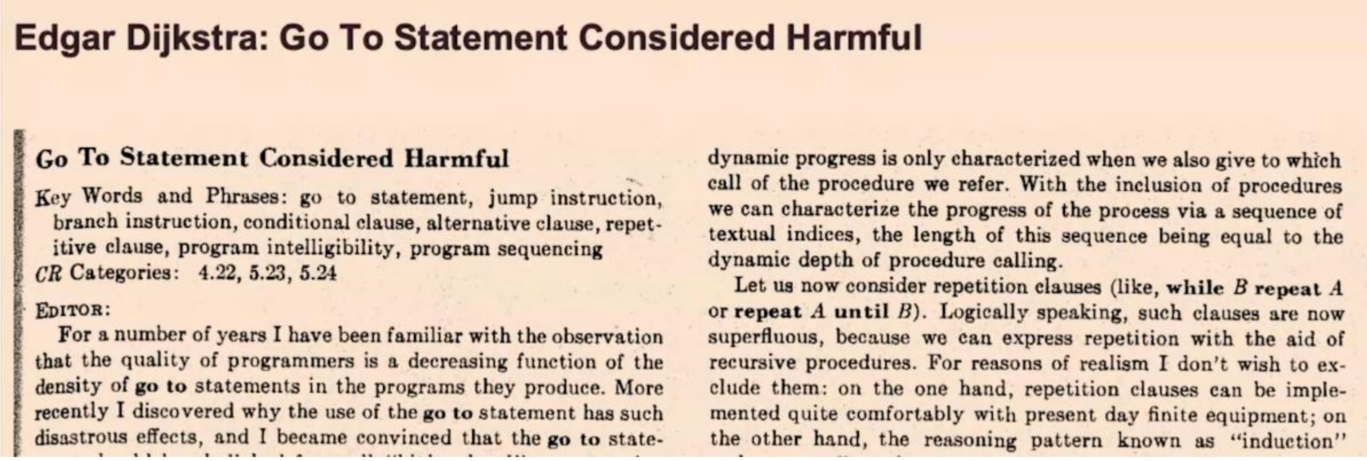
Add a ticket to restock the file cabinet

Read this email thread and update the CRM for AcmeCorp

What's the status of my package

That's wrong, use the XYZ project

Factor 4
Tools are structured outputs



Edgar Dijkstra: Go To Statement Considered Harmful

Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing
CR Categories: 4.22, 5.23, 5.24

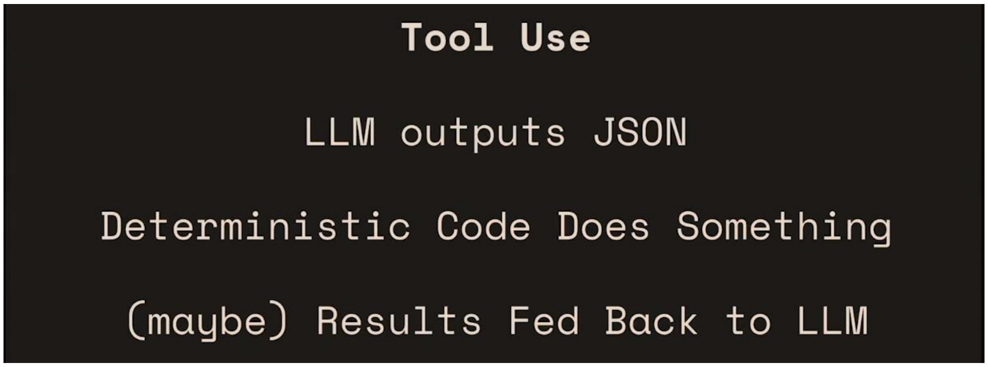
EDITOR:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** state-

dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the dynamic depth of procedure calling.

Let us now consider repetition clauses (like, **while** *B* **repeat** *A* or **repeat** *A* **until** *B*). Logically speaking, such clauses are now superfluous, because we can express repetition with the aid of recursive procedures. For reasons of realism I don't wish to exclude them: on the one hand, repetition clauses can be implemented quite comfortably with present day finite equipment; on the other hand, the reasoning pattern known as "induction"

“Tool Use” Considered Harmful

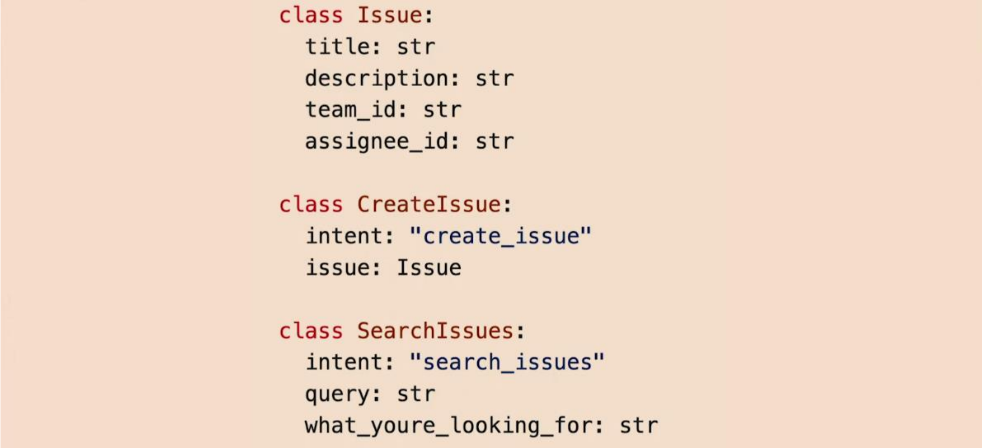


Tool Use

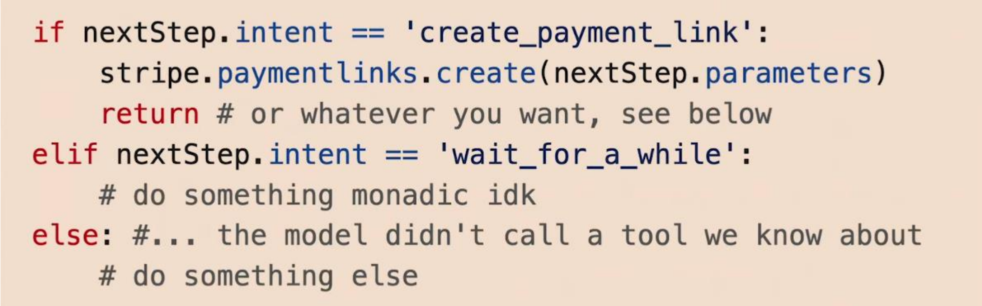
LLM outputs JSON

Deterministic Code Does Something

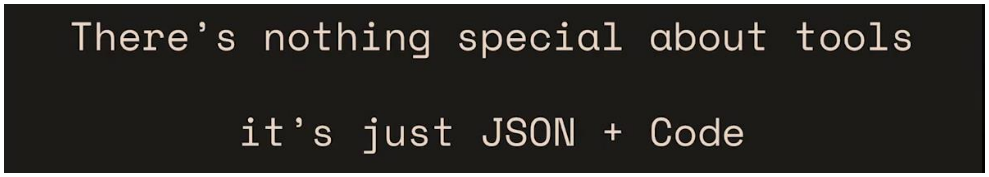
(maybe) Results Fed Back to LLM



```
class Issue:
  title: str
  description: str
  team_id: str
  assignee_id: str

class CreateIssue:
  intent: "create_issue"
  issue: Issue

class SearchIssues:
  intent: "search_issues"
  query: str
  what_youre_looking_for: str
```

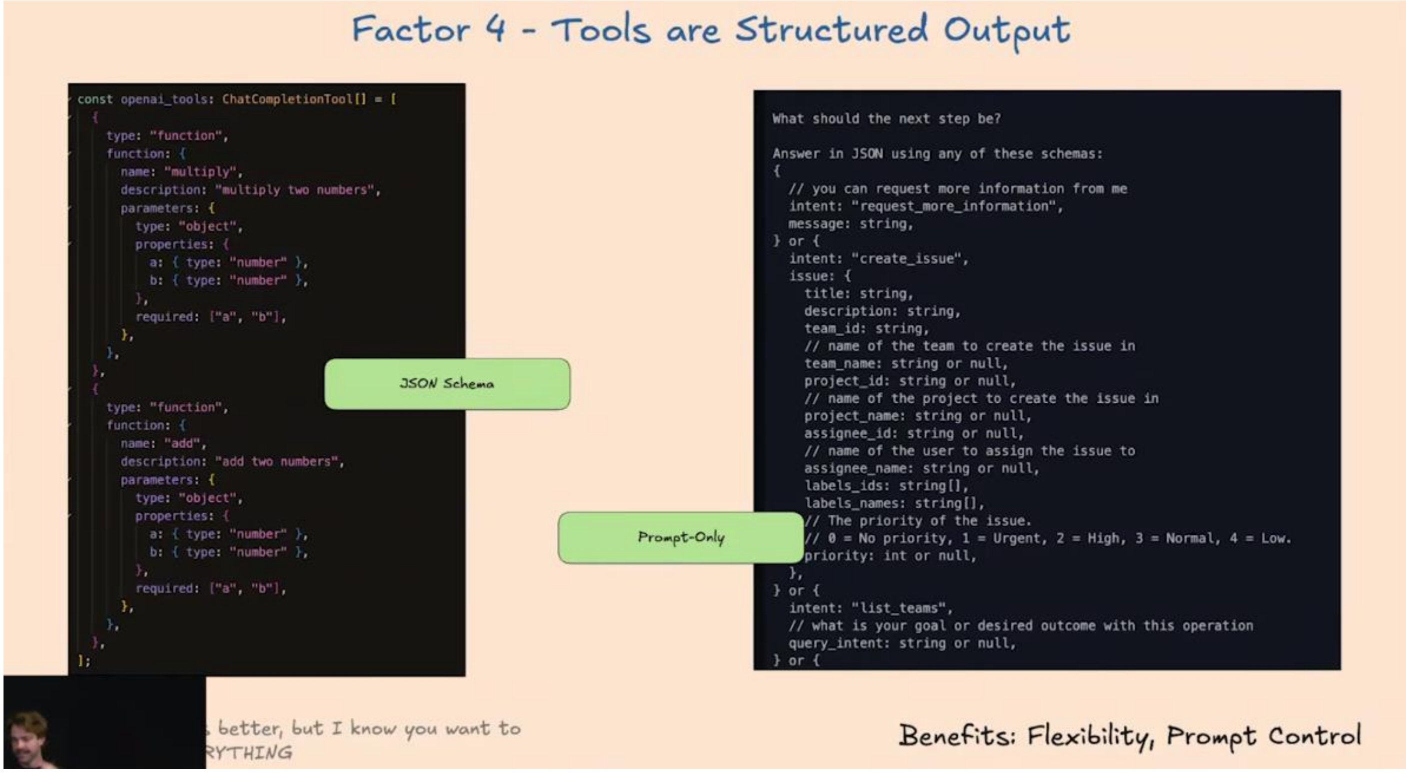


```
if nextStep.intent == 'create_payment_link':
    stripe.paymentlinks.create(nextStep.parameters)
    return # or whatever you want, see below
elif nextStep.intent == 'wait_for_a_while':
    # do something monadic idk
else: #... the model didn't call a tool we know about
    # do something else
```



There’s nothing special about tools

it’s just JSON + Code



Factor 4 - Tools are Structured Output

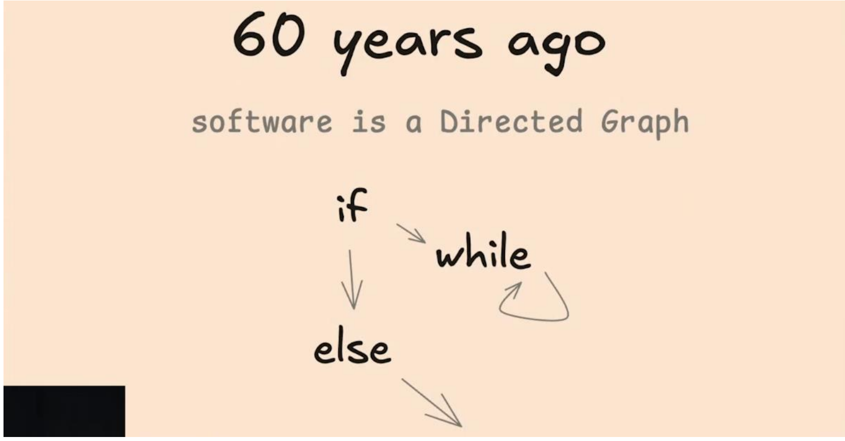
```
const openai_tools: ChatCompletionTool[] = [
  {
    type: "function",
    function: {
      name: "multiply",
      description: "multiply two numbers",
      parameters: {
        type: "object",
        properties: {
          a: { type: "number" },
          b: { type: "number" },
        },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

```
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 {
```

better, but I know you want to
RYTHING

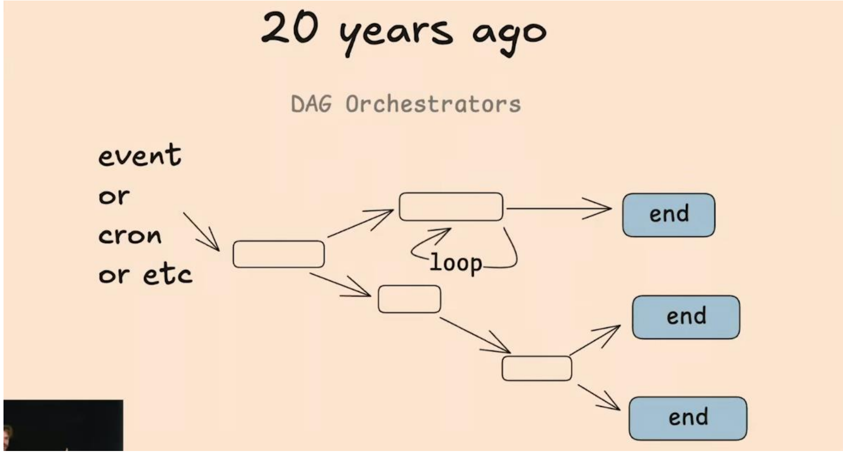
Benefits: Flexibility, Prompt Control

Factor 8
Own your control flow



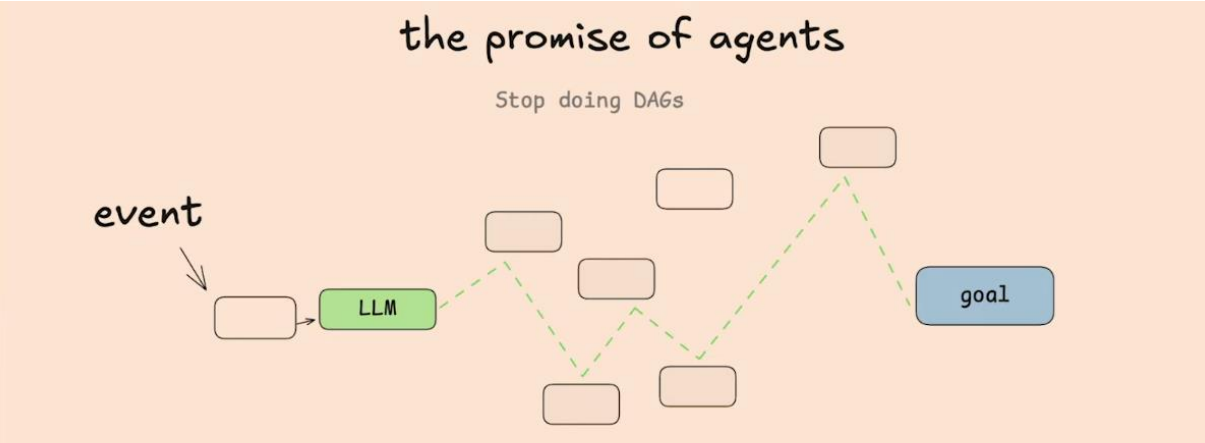
60 years ago

software is a Directed Graph



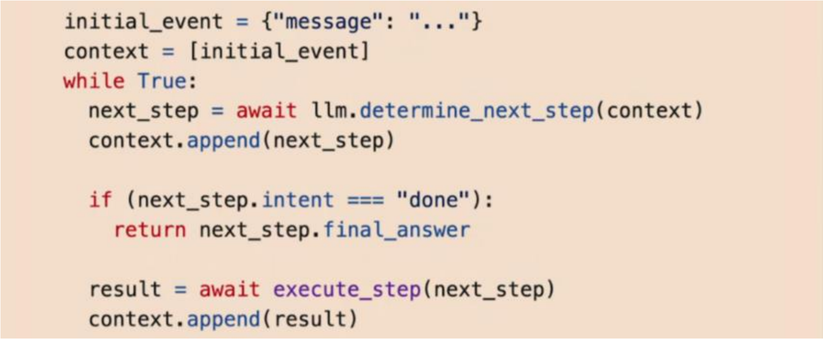
20 years ago

DAG Orchestrators

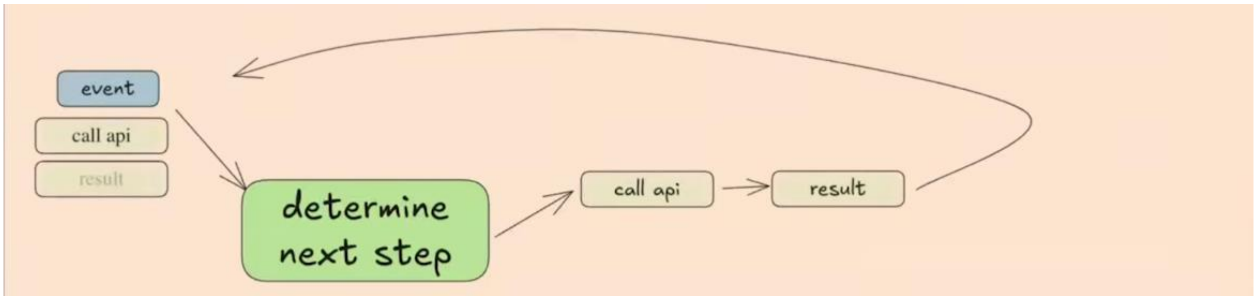


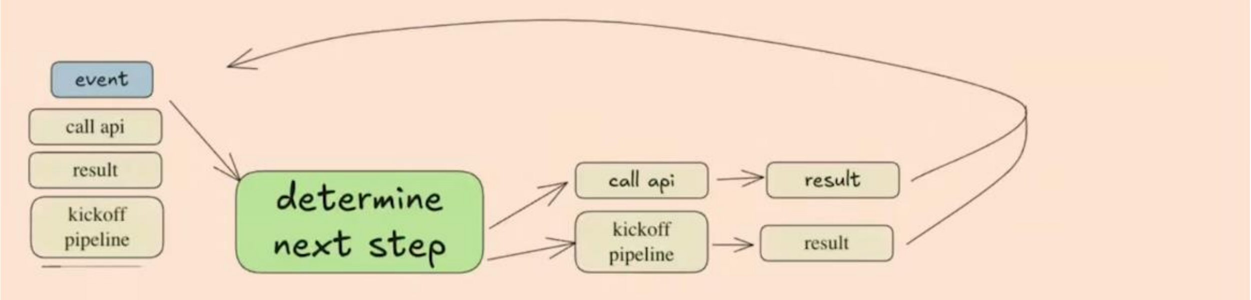
the promise of agents

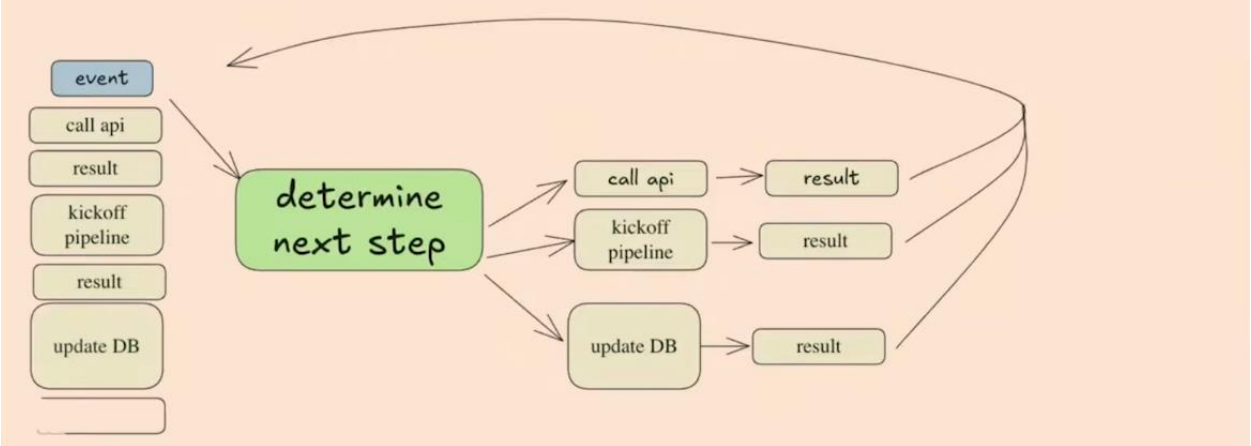
Stop doing DAGs

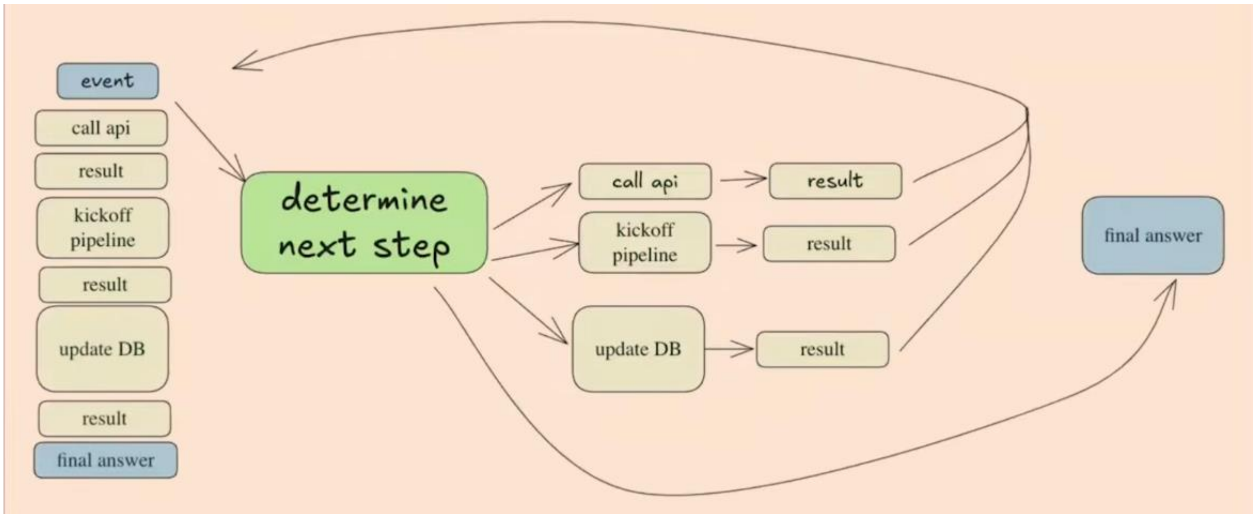


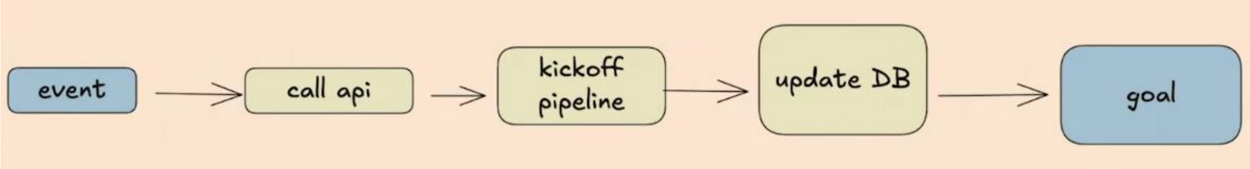
```
initial_event = {"message": "..."}
context = [initial_event]
while True:
  next_step = await llm.determine_next_step(context)
  context.append(next_step)

  if (next_step.intent === "done"):
    return next_step.final_answer

  result = await execute_step(next_step)
  context.append(result)
```





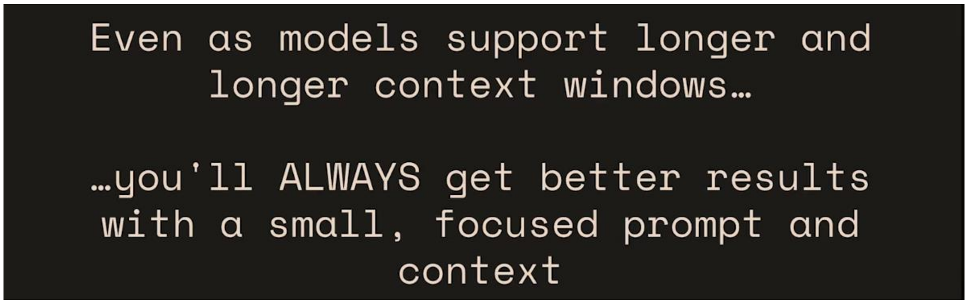






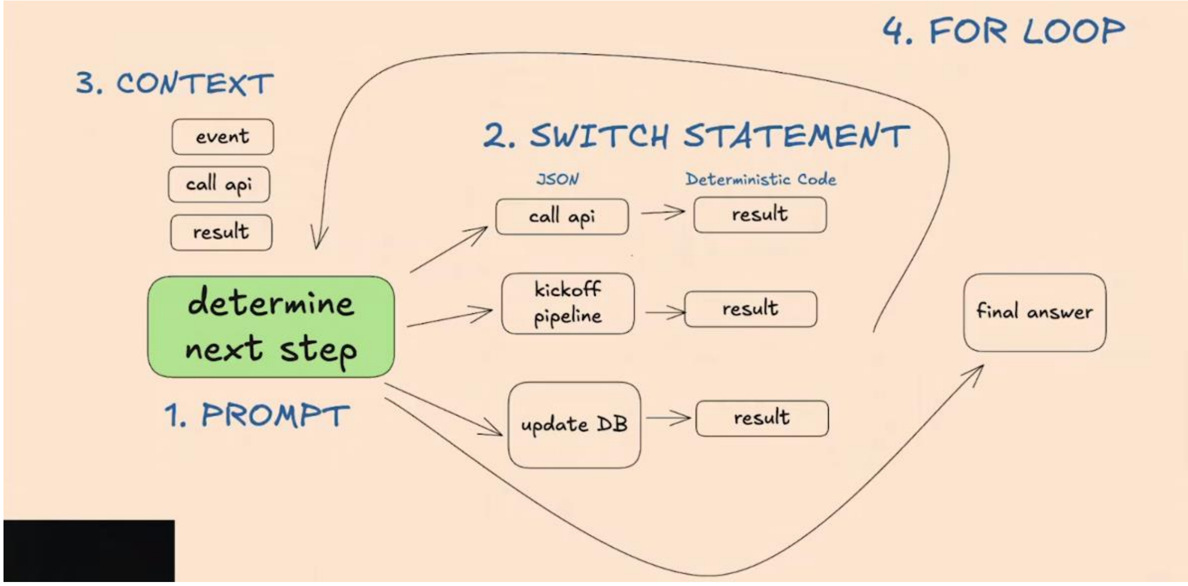
Turns out this doesn't really work

Mostly: Long Context windows

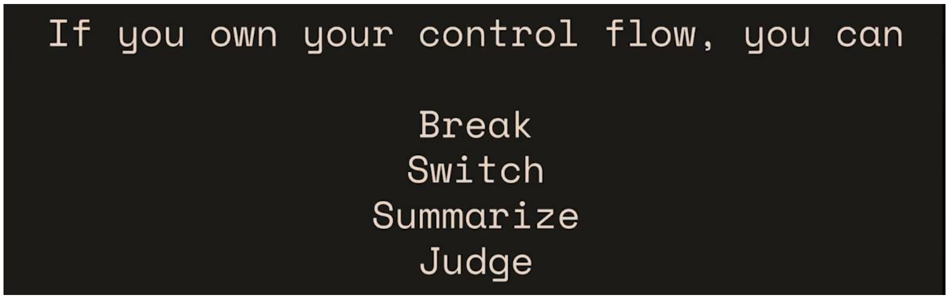

Even as models support longer and longer context windows…

…you'll ALWAYS get better results with a small, focused prompt and context

what's an agent really?!

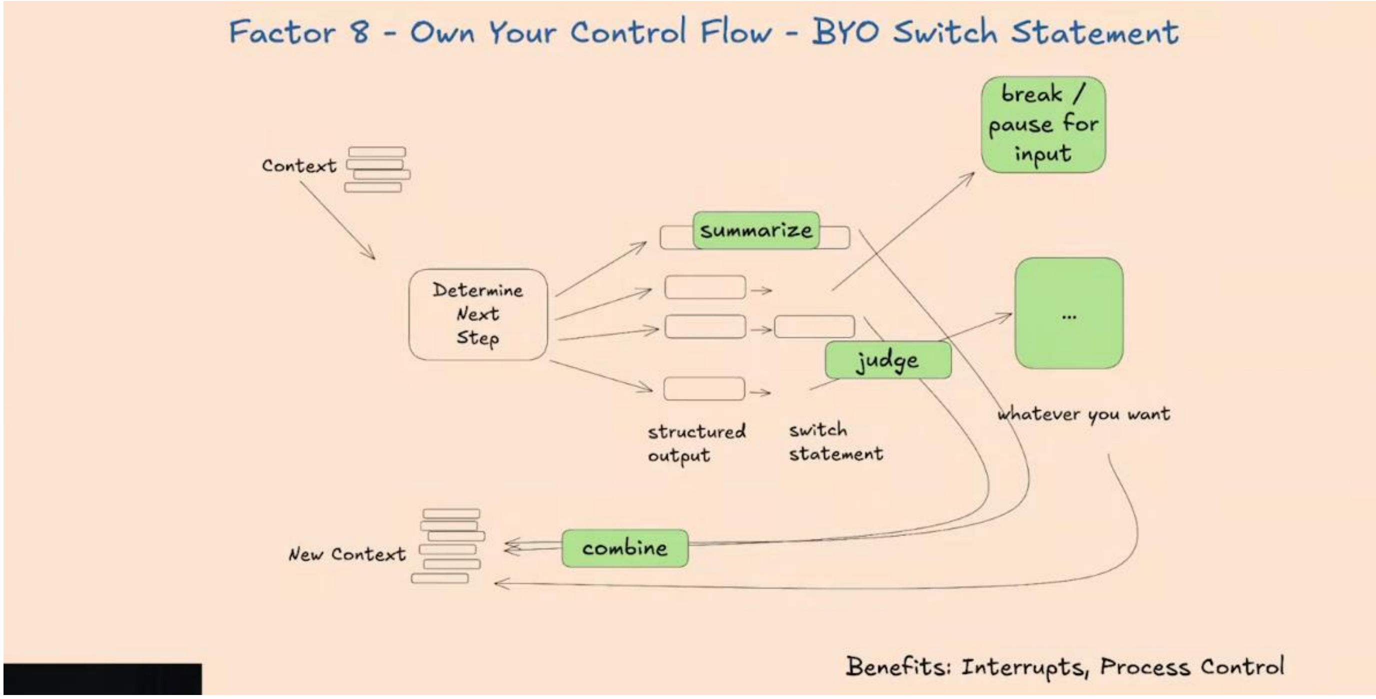


[redacted]



If you own your control flow, you can

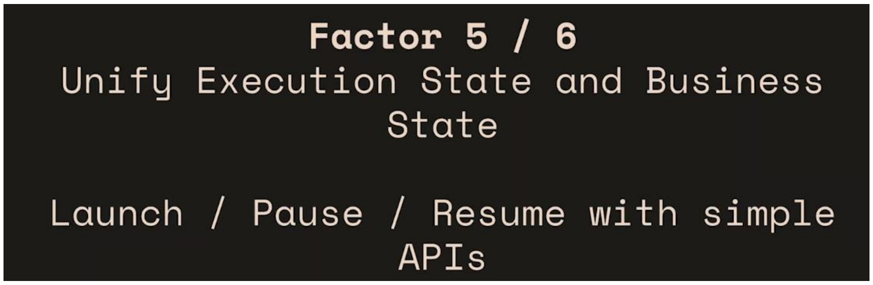
Break
Switch
Summarize
Judge



Factor 8 - Own Your Control Flow - BYO Switch Statement

[redacted]

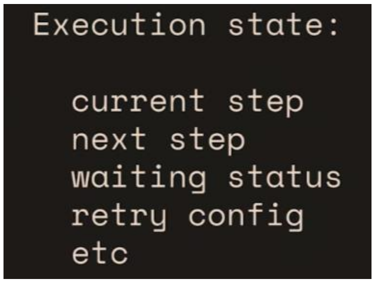
Benefits: Interrupts, Process Control



Factor 5 / 6

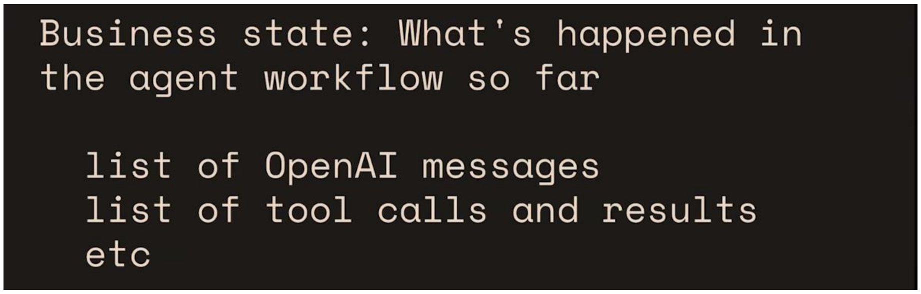
Unify Execution State and Business State

Launch / Pause / Resume with simple APIs



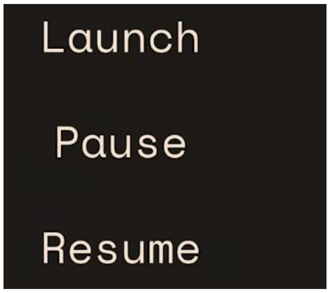
Execution state:

current step
next step
waiting status
retry config
etc



Business state: What's happened in the agent workflow so far

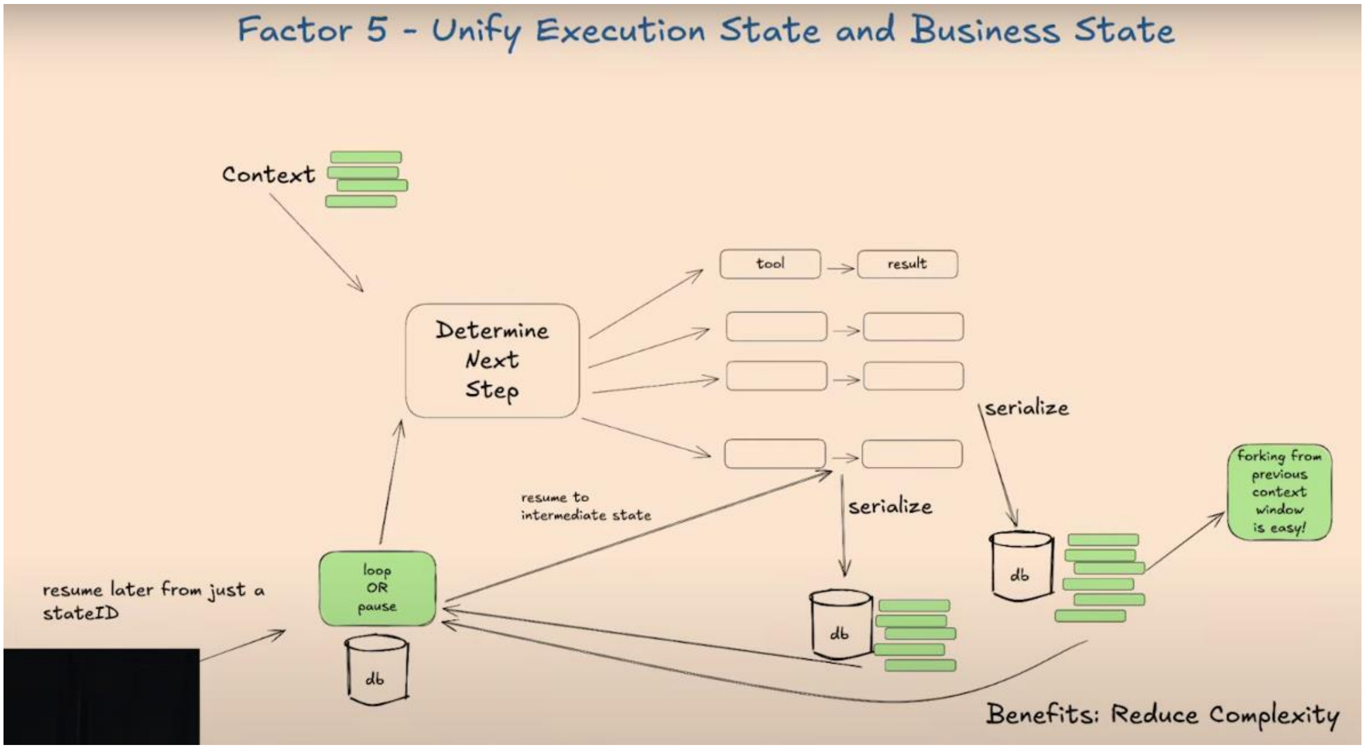
list of OpenAI messages
list of tool calls and results
etc



Launch

Pause

Resume



Factor 5 - Unify Execution State and Business State

Context

Determine Next Step

tool → result

serialize

serialize

forking from previous context window is easy!

resume to intermediate state

loop OR pause

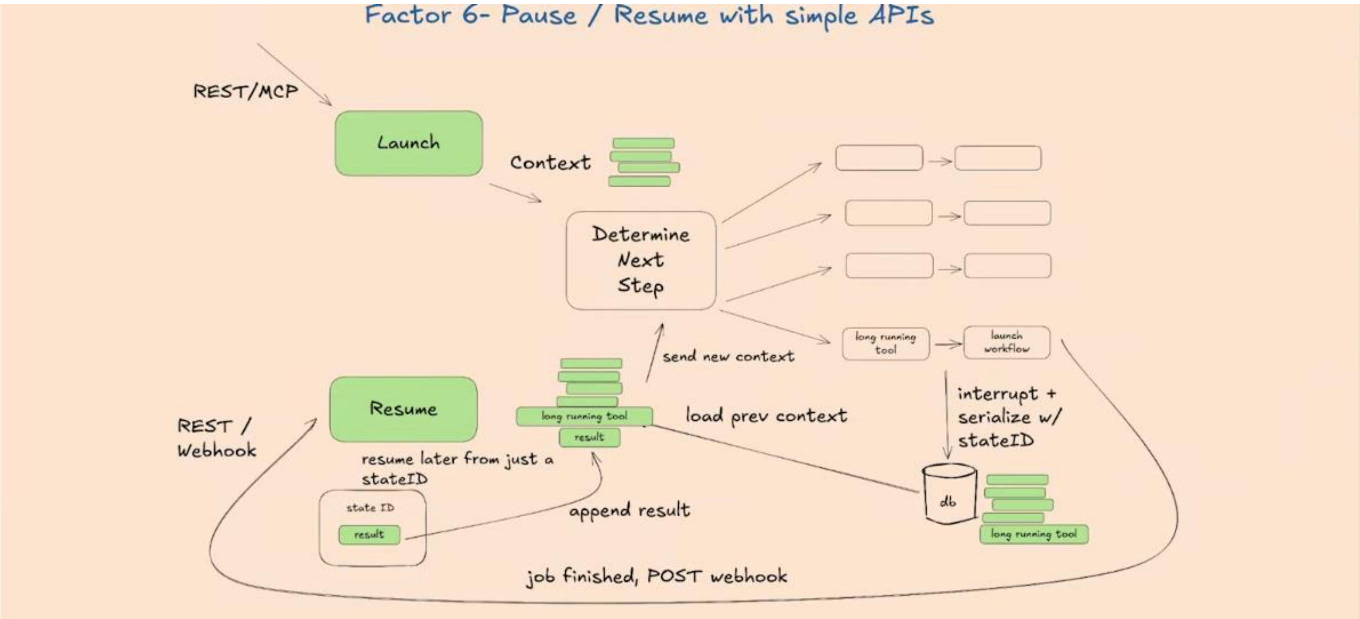
resume later from just a stateID

db

db

db

Benefits: Reduce Complexity



Factor 6- Pause / Resume with simple APIs

REST/MCP

Launch

Context

Determine Next Step

long running tool → launch workflow

send new context

interrupt + serialize w/ stateID

Resume

long running tool

result

load prev context

REST / Webhook

resume later from just a stateID

state ID

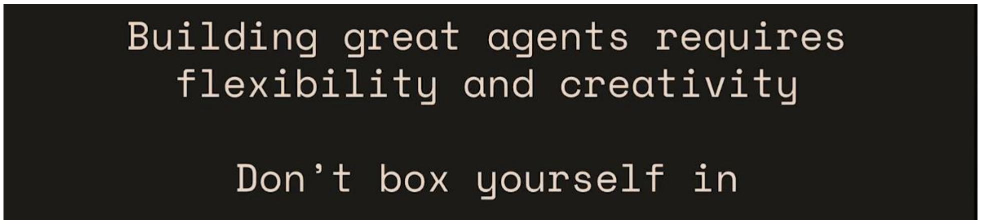
result

append result

db

long running tool

job finished, POST webhook

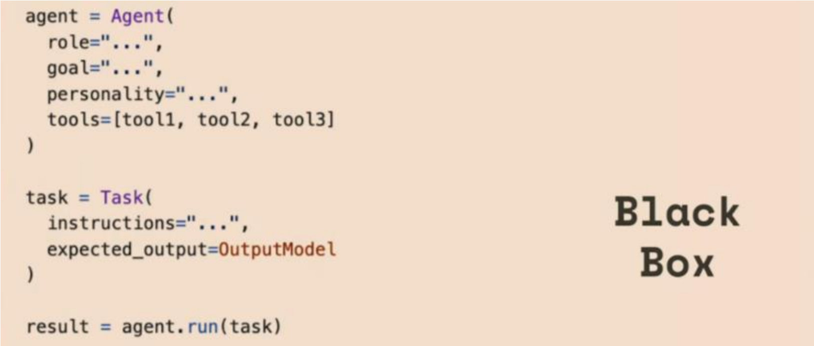


Building great agents requires flexibility and creativity

Don't box yourself in

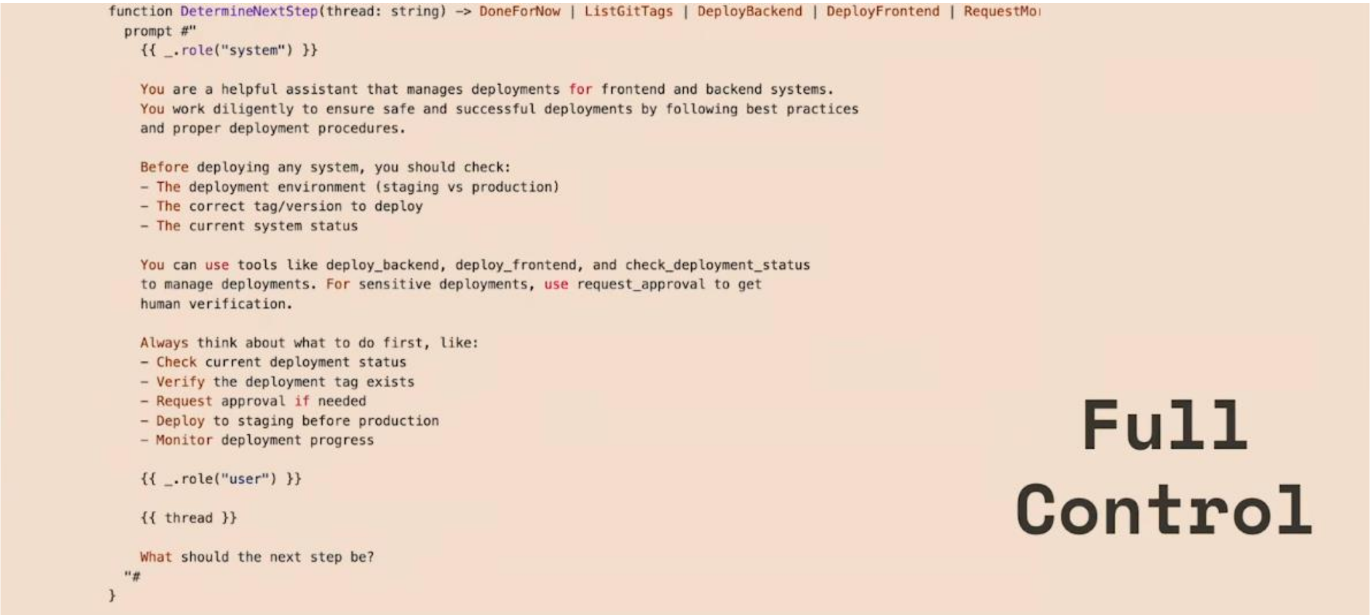
Factor 2

Own your prompts



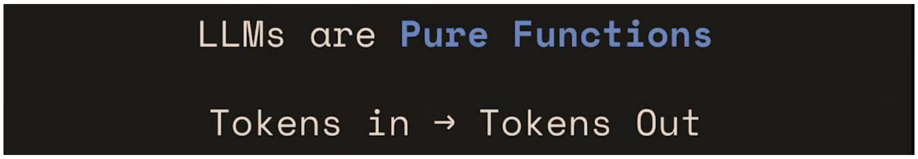
```
agent = Agent(
  role="...",
  goal="...",
  personality="...",
  tools=[tool1, tool2, tool3]
)

task = Task(
  instructions="...",
  expected_output=OutputModel
)

result = agent.run(task)
```

Black Box



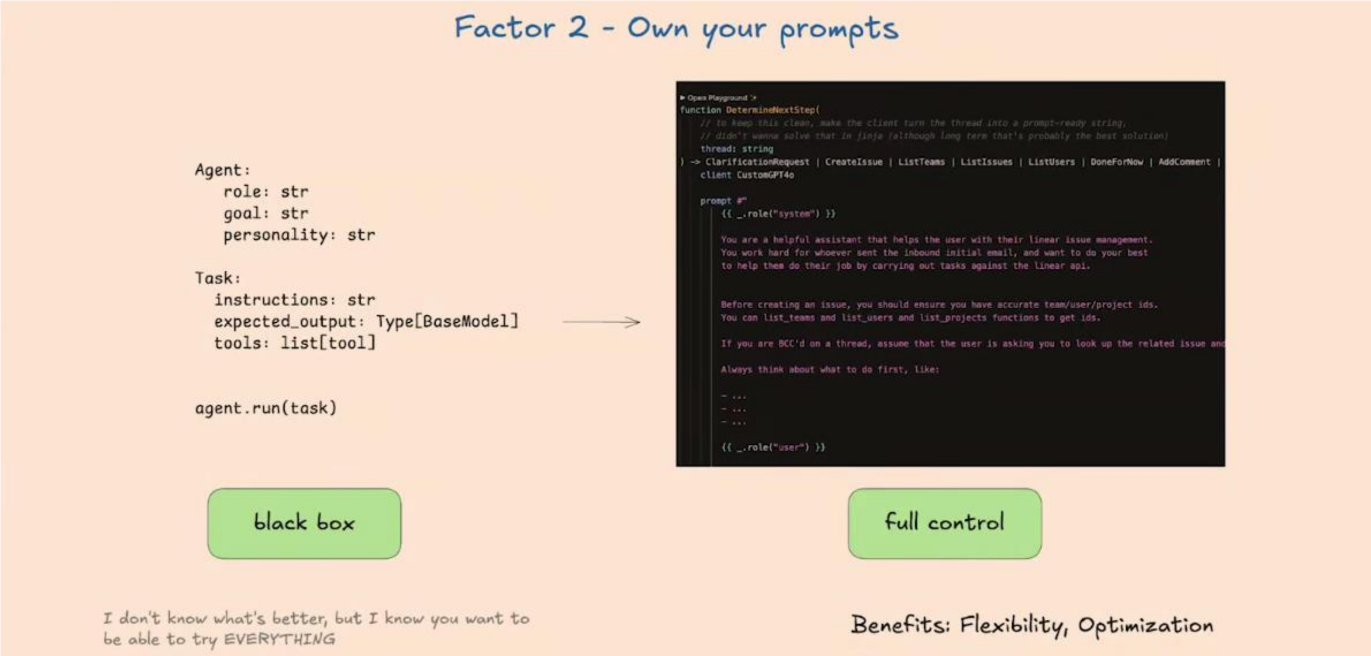
```
function DetermineNextStep(thread: string) -> DoneForNow | ListGitTags | DeployBackend | DeployFrontend | RequestMo
  prompt #"
    {{ _.role("system") }}

    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?
  "#
}
```

Full Control



LLMs are **Pure Functions**

Tokens in → Tokens Out



Factor 2 - Own your prompts

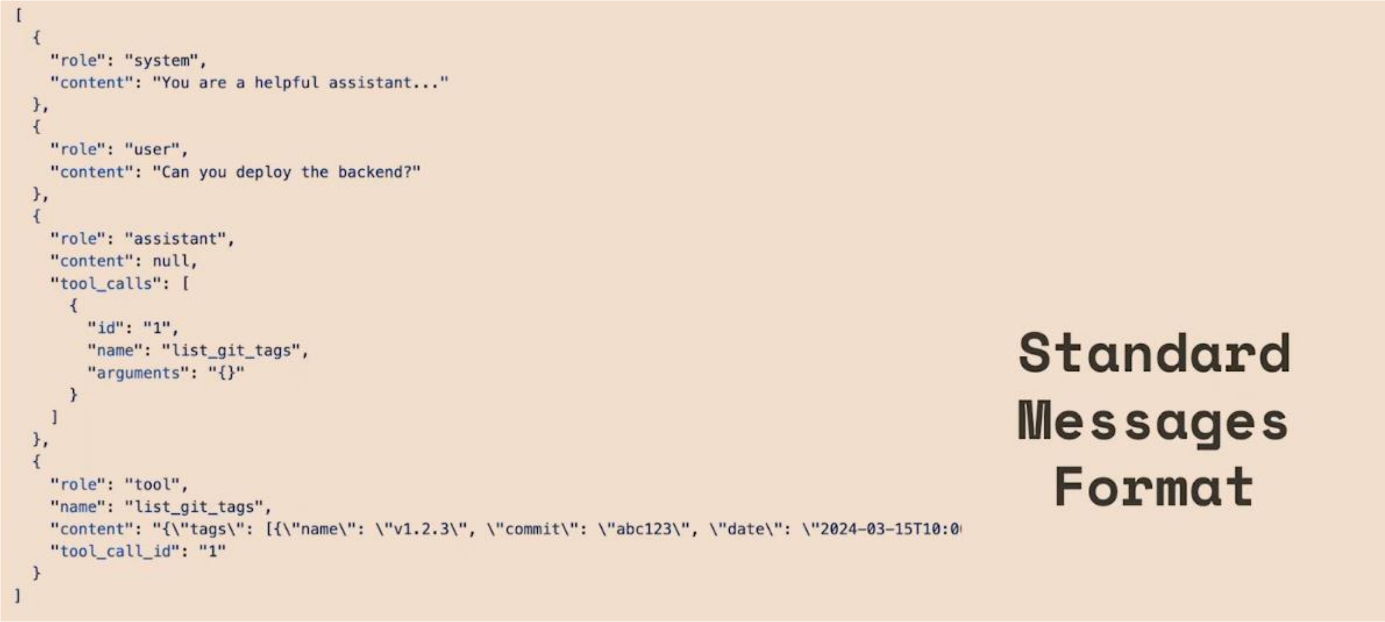
```
Agent:
   role: str
   goal: str
   personality: str

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


agent.run(task)
```

```
function DetermineNextStep(
    // to keep this clean, make the client turn the thread into a prompt-ready string,
    // didn't wanna solve that in jinja (although long term that's probably the best solution)
    thread: string
) -> ClarificationRequest | CreateIssue | ListTeams | ListIssues | ListUsers | DoneForNow | AddComment |
    client CustomGPT4o

    prompt #"
      {{ _.role("system") }}

      You are a helpful assistant that helps the user with their linear issue management.
      You work hard for whoever sent the inbound initial email, and want to do your best
      to help them do their job by carrying out tasks against the linear api.


      Before creating an issue, you should ensure you have accurate team/user/project ids.
      You can list_teams and list_users and list_projects functions to get ids.

      If you are BCC'd on a thread, assume that the user is asking you to look up the related issue and

      Always think about what to do first, like:

      - ...
      - ...
      - ...

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

I don't know what's better, but I know you want to be able to try EVERYTHING

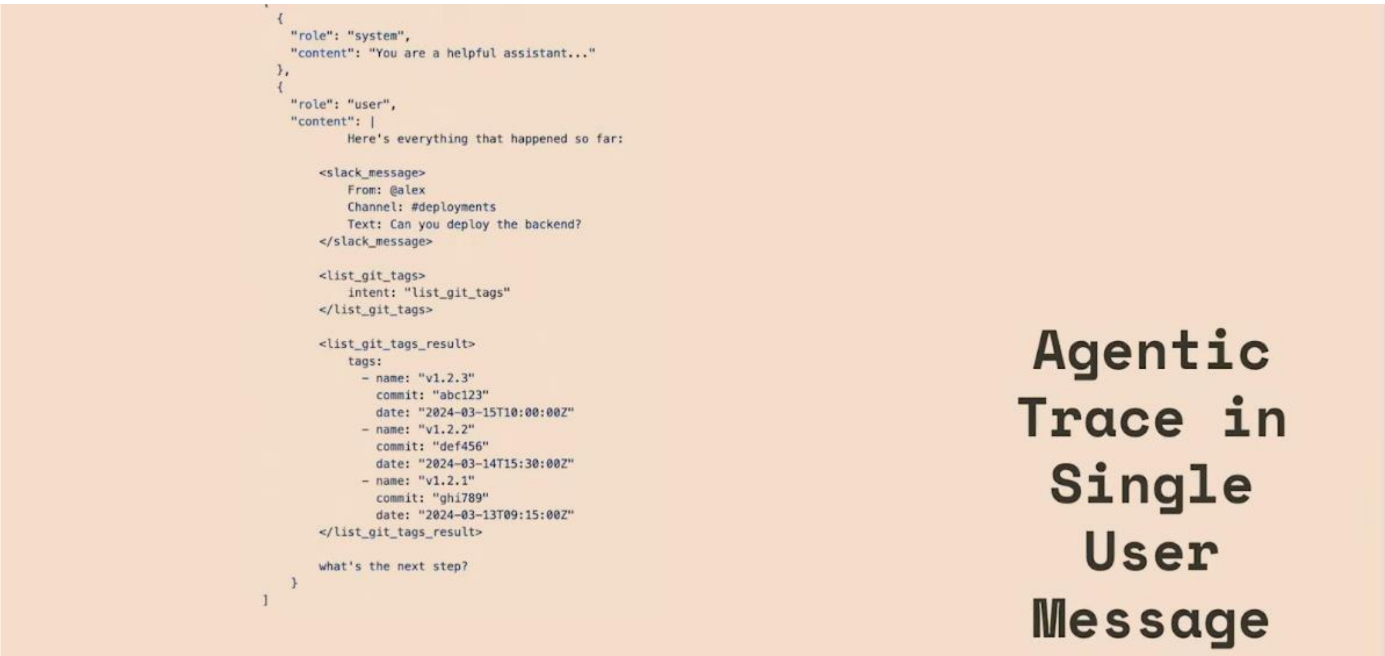
Benefits: Flexibility, Optimization

Factor 3
Own your Context Building



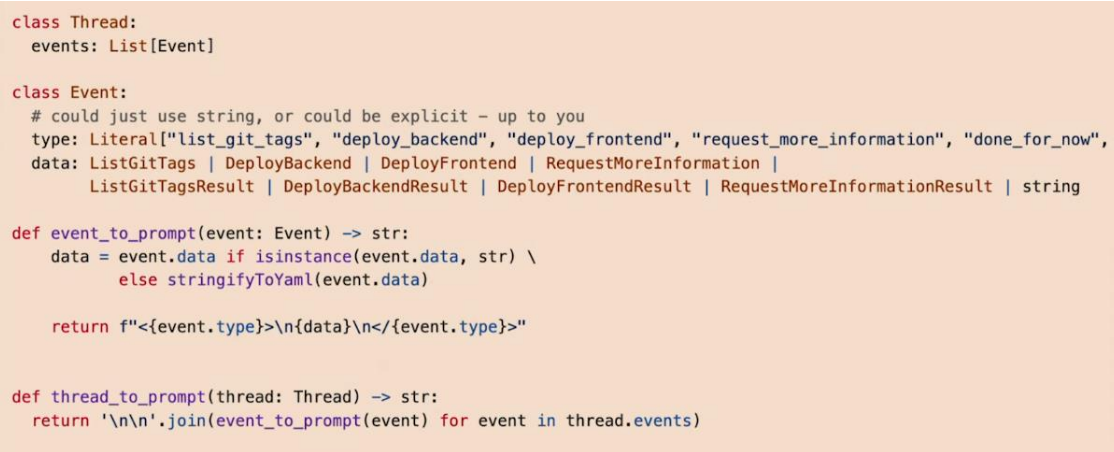
```
[
  {
    "role": "system",
    "content": "You are a helpful assistant..."
  },
  {
    "role": "user",
    "content": "Can you deploy the backend?"
  },
  {
    "role": "assistant",
    "content": null,
    "tool_calls": [
      {
        "id": "1",
        "name": "list_git_tags",
        "arguments": "{}"
      }
    ]
  },
  {
    "role": "tool",
    "name": "list_git_tags",
    "content": "{\"tags\": [{\"name\": \"v1.2.3\", \"commit\": \"abc123\", \"date\": \"2024-03-15T10:0
    "tool_call_id": "1"
  }
]
```

Standard Messages Format

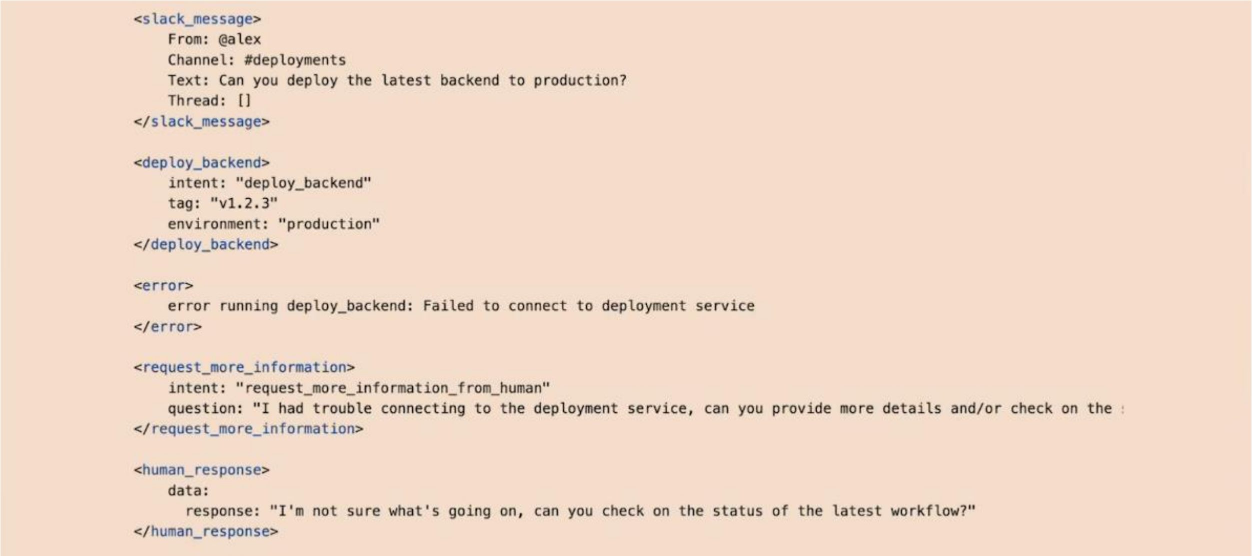


```
  {
    "role": "system",
    "content": "You are a helpful assistant..."
  },
  {
    "role": "user",
    "content": |
            Here's everything that happened so far:

        <slack_message>
            From: @alex
            Channel: #deployments
            Text: Can you deploy the backend?
        </slack_message>

        <list_git_tags>
            intent: "list_git_tags"
        </list_git_tags>

        <list_git_tags_result>
            tags:
              - name: "v1.2.3"
                commit: "abc123"
                date: "2024-03-15T10:00:00Z"
              - name: "v1.2.2"
                commit: "def456"
                date: "2024-03-14T15:30:00Z"
              - name: "v1.2.1"
                commit: "ghi789"
                date: "2024-03-13T09:15:00Z"
        </list_git_tags_result>

        what's the next step?
    }
]
```

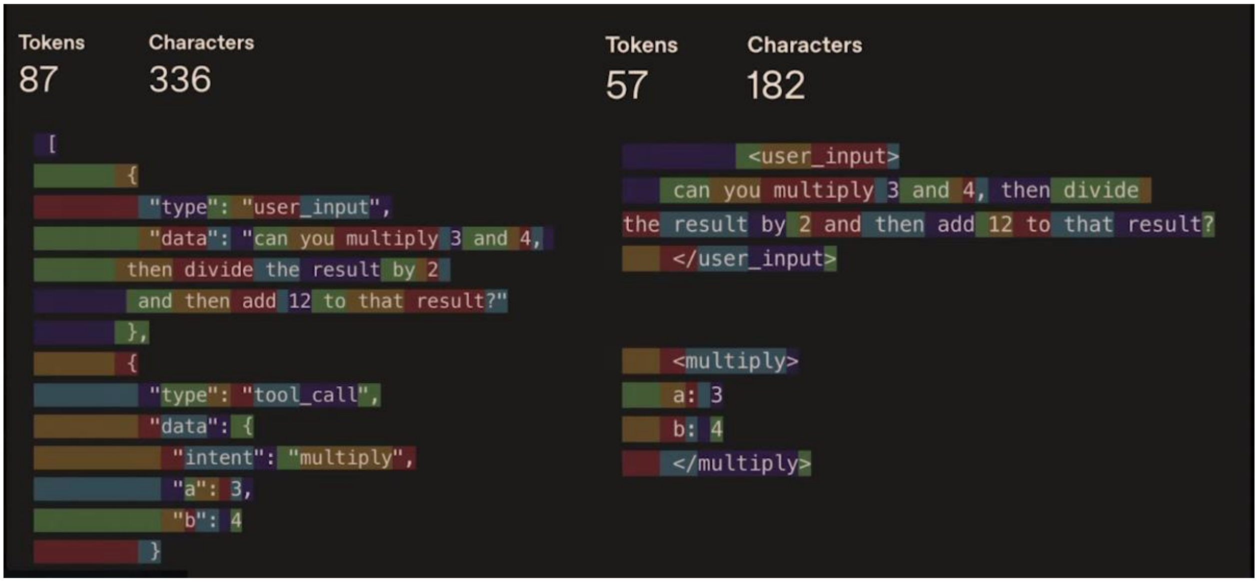
Agentic Trace in Single User Message



```
class Thread:
  events: List[Event]

class Event:
  # could just use string, or could be explicit - up to you
  type: Literal["list_git_tags", "deploy_backend", "deploy_frontend", "request_more_information", "done_for_now",
  data: ListGitTags | DeployBackend | DeployFrontend | RequestMoreInformation |
        ListGitTagsResult | DeployBackendResult | DeployFrontendResult | RequestMoreInformationResult | string

def event_to_prompt(event: Event) -> str:
    data = event.data if isinstance(event.data, str) \
           else stringifyToYaml(event.data)

    return f"<{event.type}>\n{data}\n</{event.type}>"


def thread_to_prompt(thread: Thread) -> str:
  return '\n\n'.join(event_to_prompt(event) for event in thread.events)
```



```
<slack_message>
    From: @alex
    Channel: #deployments
    Text: Can you deploy the latest backend to production?
    Thread: []
</slack_message>

<deploy_backend>
    intent: "deploy_backend"
    tag: "v1.2.3"
    environment: "production"
</deploy_backend>

<error>
    error running deploy_backend: Failed to connect to deployment service
</error>

<request_more_information>
    intent: "request_more_information_from_human"
    question: "I had trouble connecting to the deployment service, can you provide more details and/or check on the
</request_more_information>

<human_response>
    data:
      response: "I'm not sure what's going on, can you check on the status of the latest workflow?"
</human_response>
```

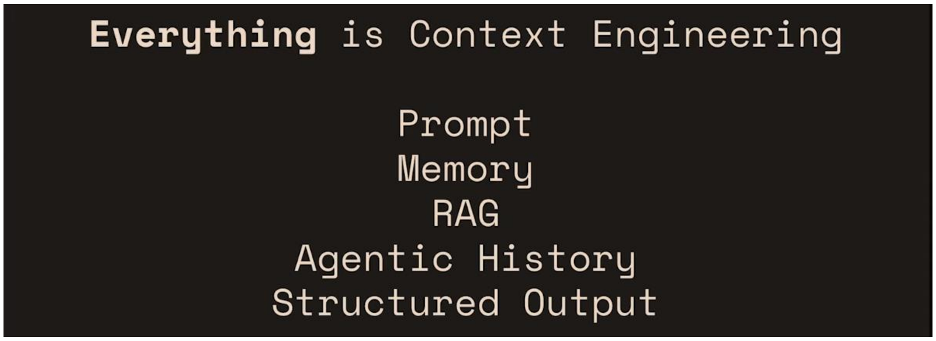


Tokens 87 Characters 336

```
[
  {
    "type": "user_input",
    "data": "can you multiply 3 and 4,
  then divide the result by 2
   and then add 12 to that result?"
  },
  {
    "type": "tool_call",
    "data": {
      "intent": "multiply",
      "a": 3,
      "b": 4
    }
```

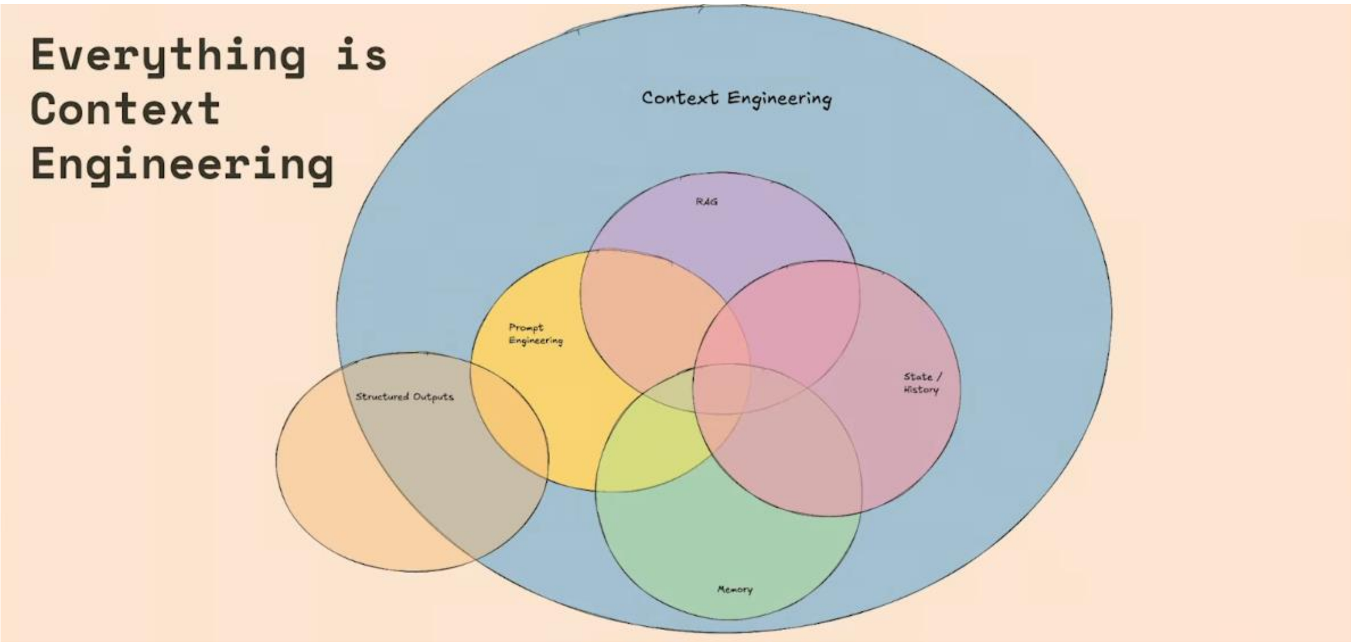
Tokens 57 Characters 182

```
      <user_input>
  can you multiply 3 and 4, then divide
the result by 2 and then add 12 to that result?
  </user_input>


  <multiply>
  a: 3
  b: 4
  </multiply>
```

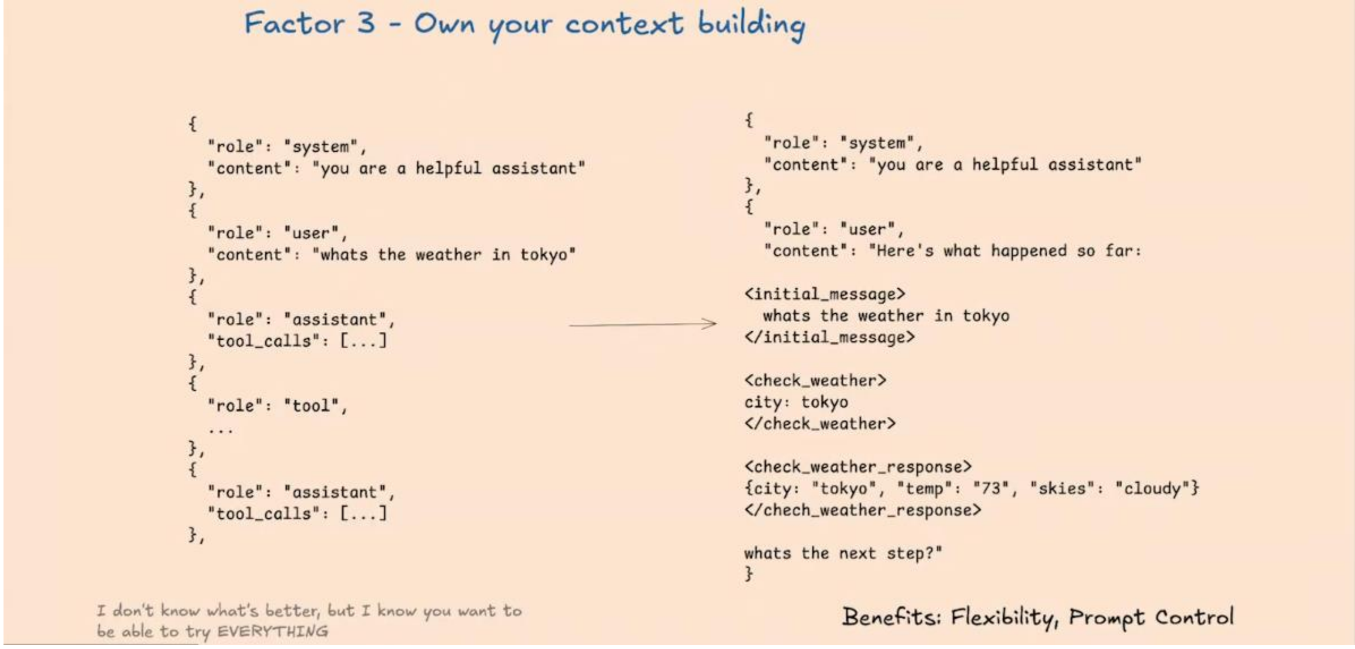


Everything is Context Engineering

Prompt
Memory
RAG
Agentic History
Structured Output



Everything is Context Engineering



Factor 3 - Own your context building

```
{
  "role": "system",
  "content": "you are a helpful assistant"
},
{
  "role": "user",
  "content": "whats the weather in tokyo"
},
{
  "role": "assistant",
  "tool_calls": [...]
},
{
  "role": "tool",
  ...
},
{
  "role": "assistant",
  "tool_calls": [...]
},
```

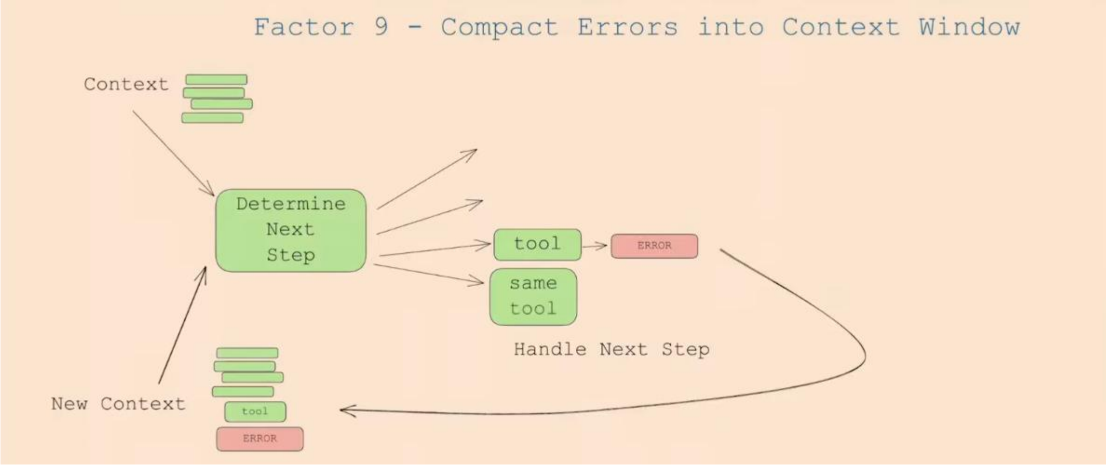
→

```
{
  "role": "system",
  "content": "you are a helpful assistant"
},
{
  "role": "user",
  "content": "Here's what happened so far:

<initial_message>
  whats the weather in tokyo
</initial_message>

<check_weather>
city: tokyo
</check_weather>

<check_weather_response>
{city: "tokyo", "temp": "73", "skies": "cloudy"}
</chech_weather_response>

whats the next step?"
}
```

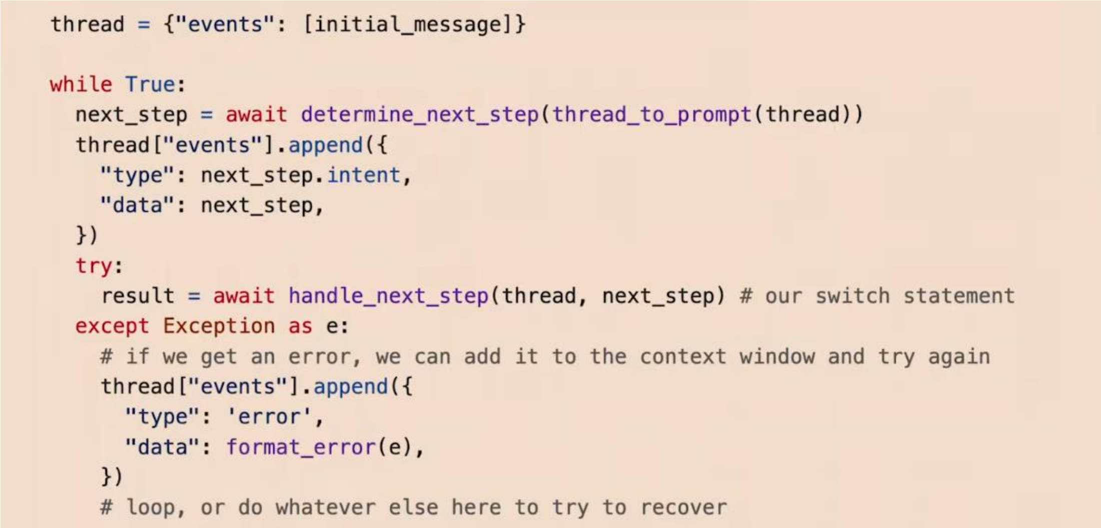
I don't know what's better, but I know you want to be able to try EVERYTHING

Benefits: Flexibility, Prompt Control

Factor 9
Compact Errors into context window

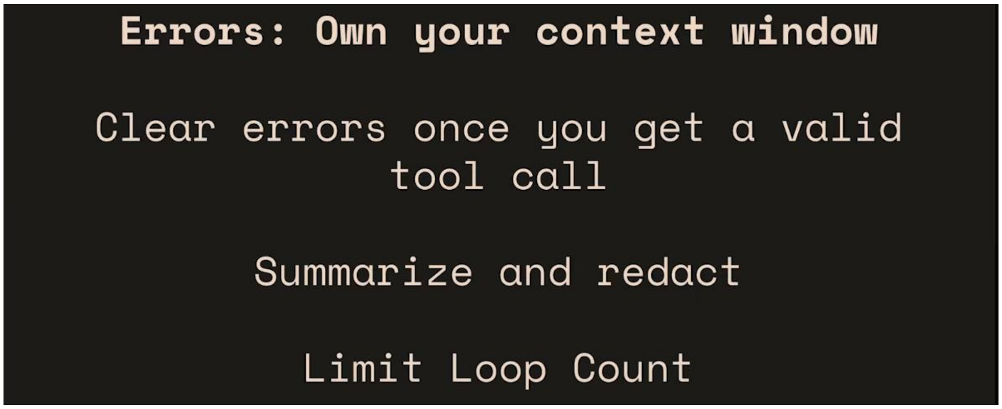


Factor 9 - Compact Errors into Context Window



```
thread = {"events": [initial_message]}

while True:
  next_step = await determine_next_step(thread_to_prompt(thread))
  thread["events"].append({
    "type": next_step.intent,
    "data": next_step,
  })
  try:
    result = await handle_next_step(thread, next_step) # our switch statement
  except Exception as e:
    # if we get an error, we can add it to the context window and try again
    thread["events"].append({
      "type": 'error',
      "data": format_error(e),
    })
    # loop, or do whatever else here to try to recover
```

What about Spin-Outs

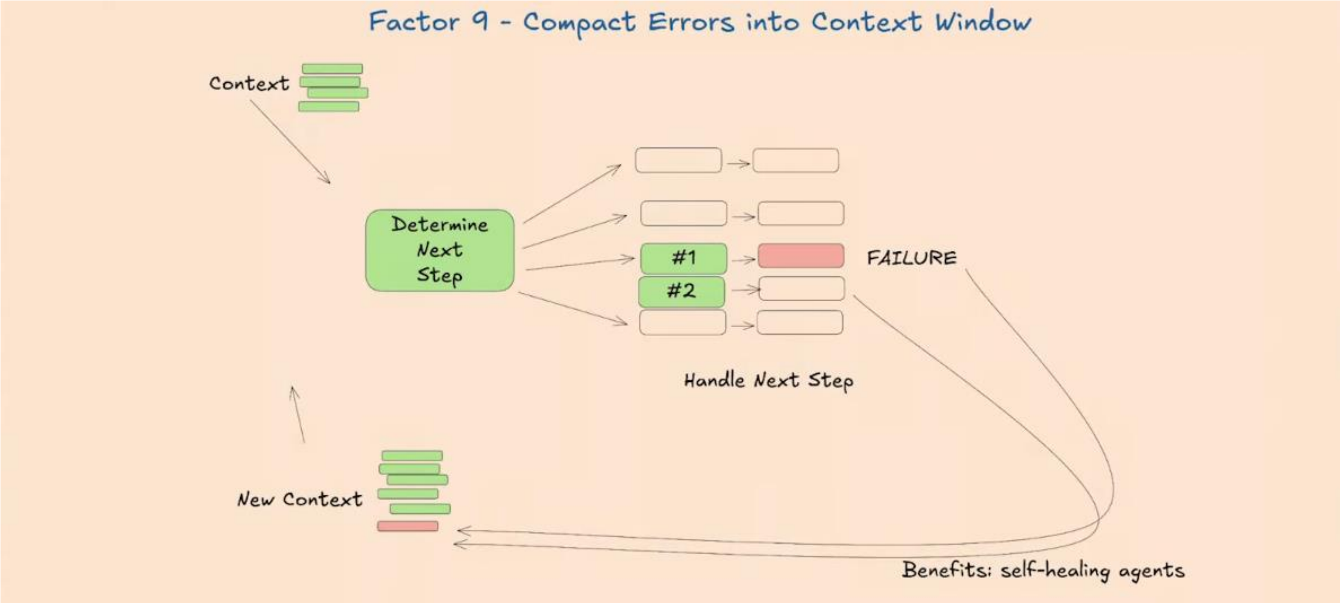


Errors: Own your context window

Clear errors once you get a valid tool call

Summarize and redact

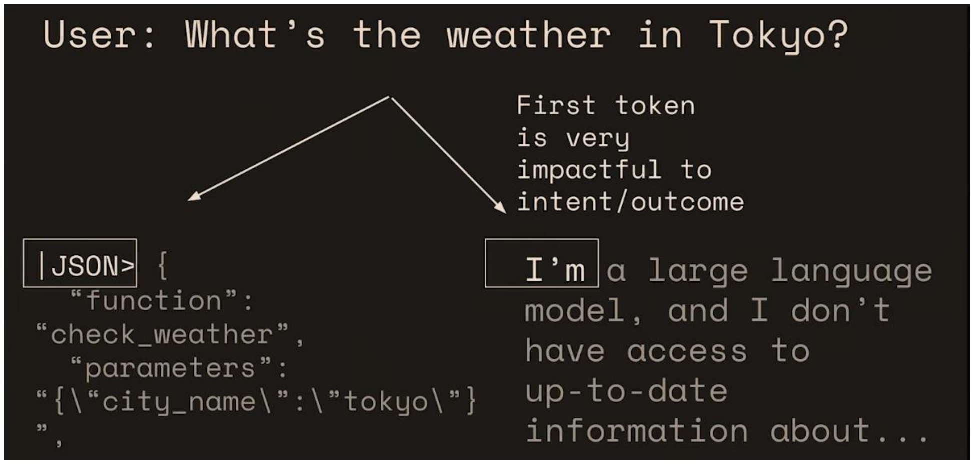
Limit Loop Count



Factor 9 - Compact Errors into Context Window

Factor 7

Contact Humans with Tools

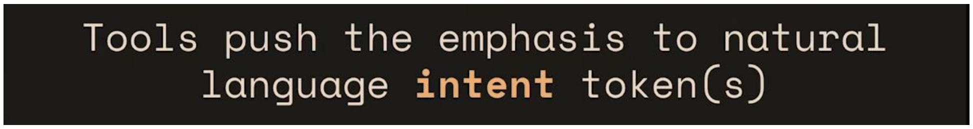


User: What’s the weather in Tokyo?

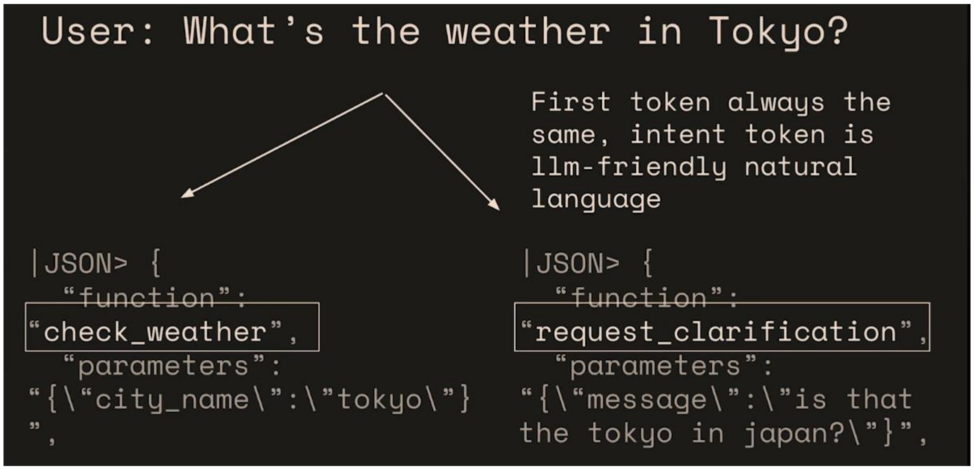
First token is very impactful to intent/outcome

|JSON> {
“function”:
“check_weather”,
“parameters”:
“{\“city_name\”:\”tokyo\”}
”,

I’m a large language model, and I don’t have access to up-to-date information about...



Tools push the emphasis to natural language **intent** token(s)

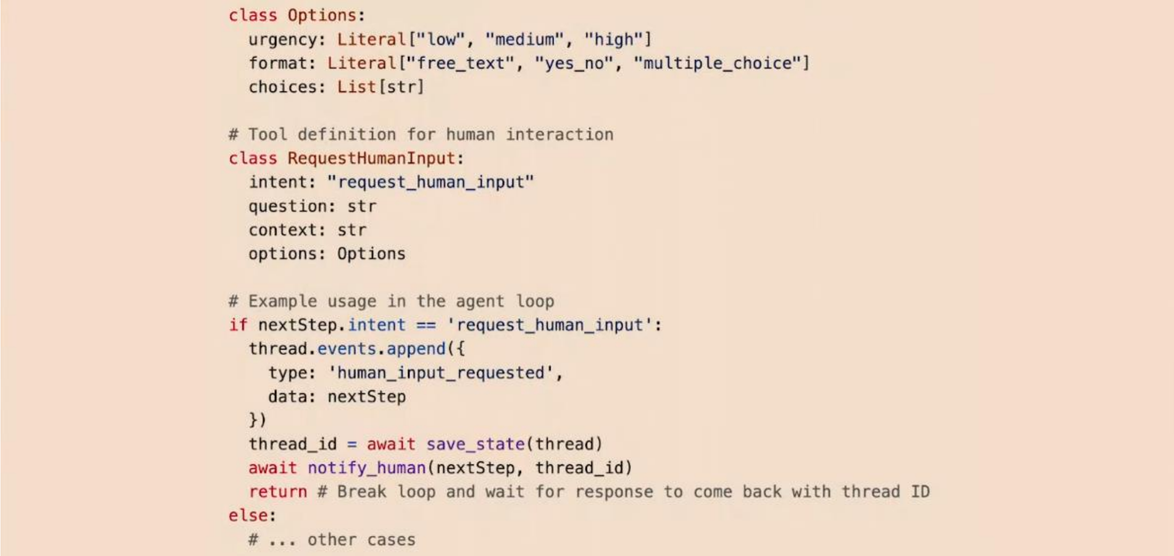


User: What’s the weather in Tokyo?

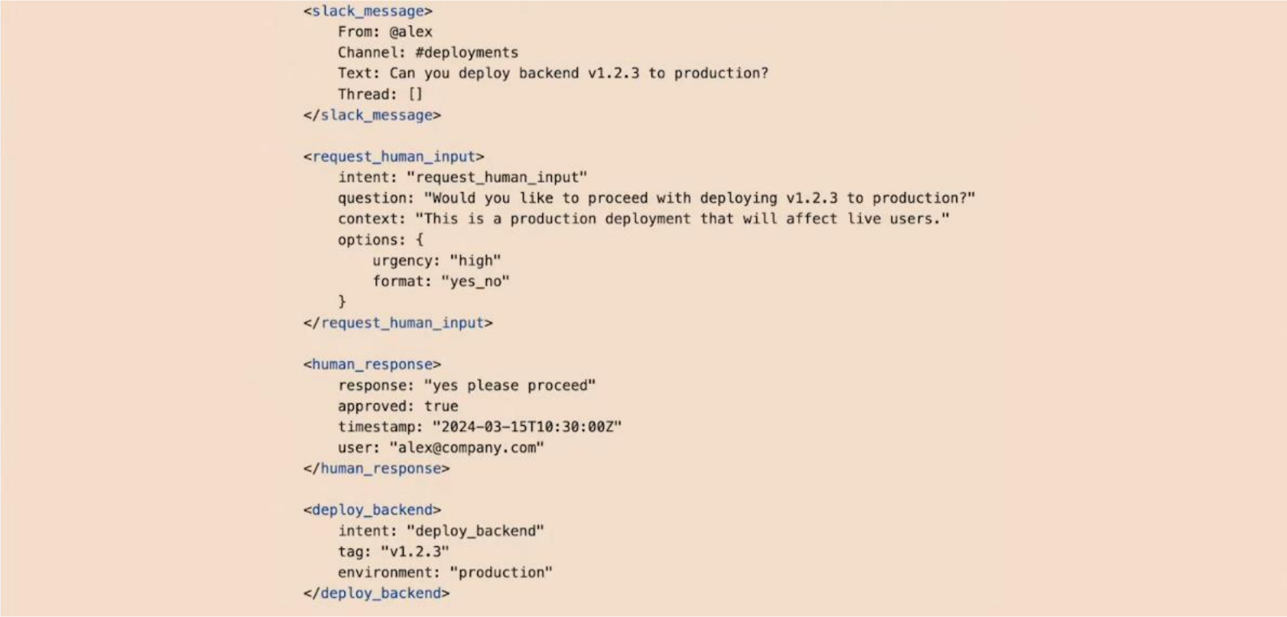
First token always the same, intent token is llm-friendly natural language

|JSON> {
“function”:
“check_weather”,
“parameters”:
“{\“city_name\”:\”tokyo\”}
”,

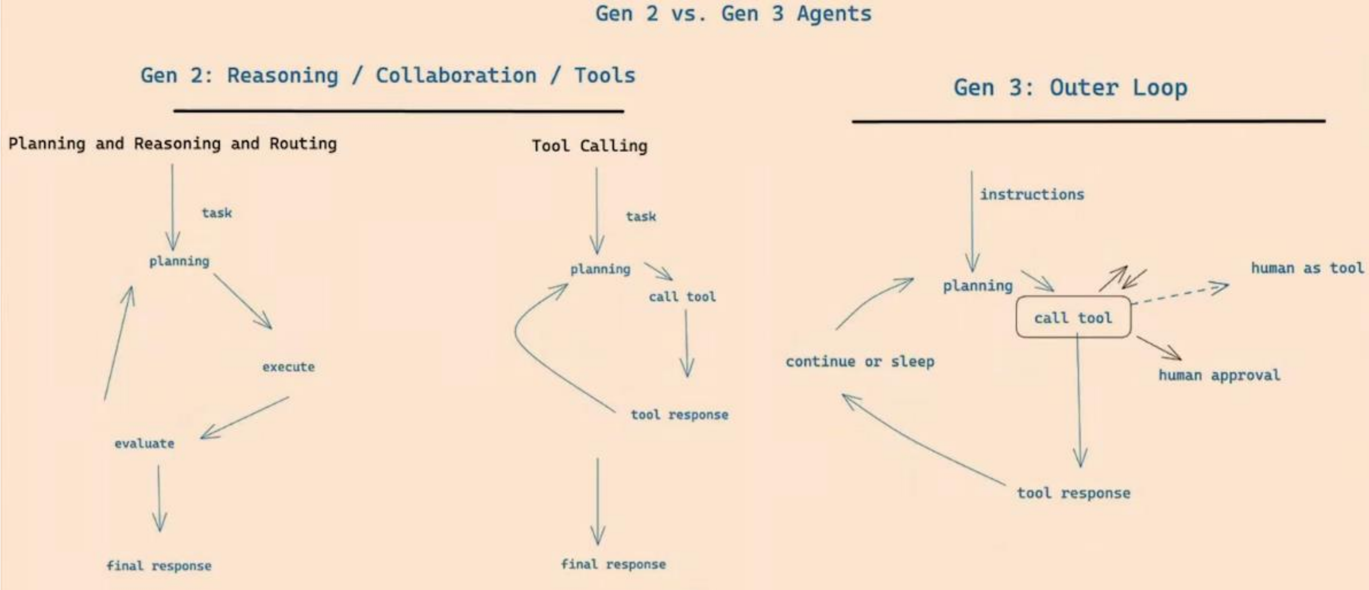
|JSON> {
“function”:
“request_clarification”,
“parameters”:
“{\“message\”:\”is that the tokyo in japan?\”}”,

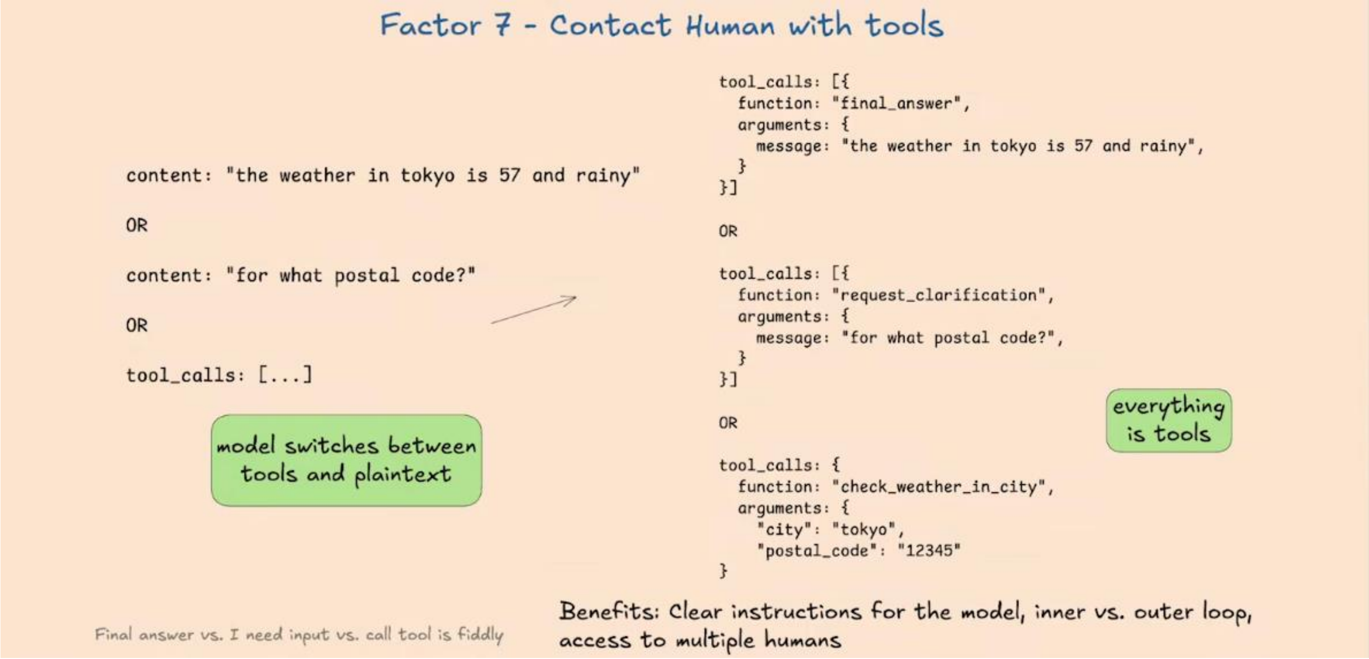


```
class Options:
  urgency: Literal["low", "medium", "high"]
  format: Literal["free_text", "yes_no", "multiple_choice"]
  choices: List[str]

# Tool definition for human interaction
class RequestHumanInput:
  intent: "request_human_input"
  question: str
  context: str
  options: Options

# Example usage in the agent loop
if nextStep.intent == 'request_human_input':
  thread.events.append({
    type: 'human_input_requested',
    data: nextStep
  })
  thread_id = await save_state(thread)
  await notify_human(nextStep, thread_id)
  return # Break loop and wait for response to come back with thread ID
else:
  # ... other cases
```



```
<slack_message>
    From: @alex
    Channel: #deployments
    Text: Can you deploy backend v1.2.3 to production?
    Thread: []
</slack_message>

<request_human_input>
    intent: "request_human_input"
    question: "Would you like to proceed with deploying v1.2.3 to production?"
    context: "This is a production deployment that will affect live users."
    options: {
        urgency: "high"
        format: "yes_no"
    }
</request_human_input>

<human_response>
    response: "yes please proceed"
    approved: true
    timestamp: "2024-03-15T10:30:00Z"
    user: "alex@company.com"
</human_response>

<deploy_backend>
    intent: "deploy_backend"
    tag: "v1.2.3"
    environment: "production"
</deploy_backend>
```





Factor 7 - Contact Human with tools

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

OR

content: "for what postal code?"

OR

tool_calls: [...]
```

model switches between tools and plaintext

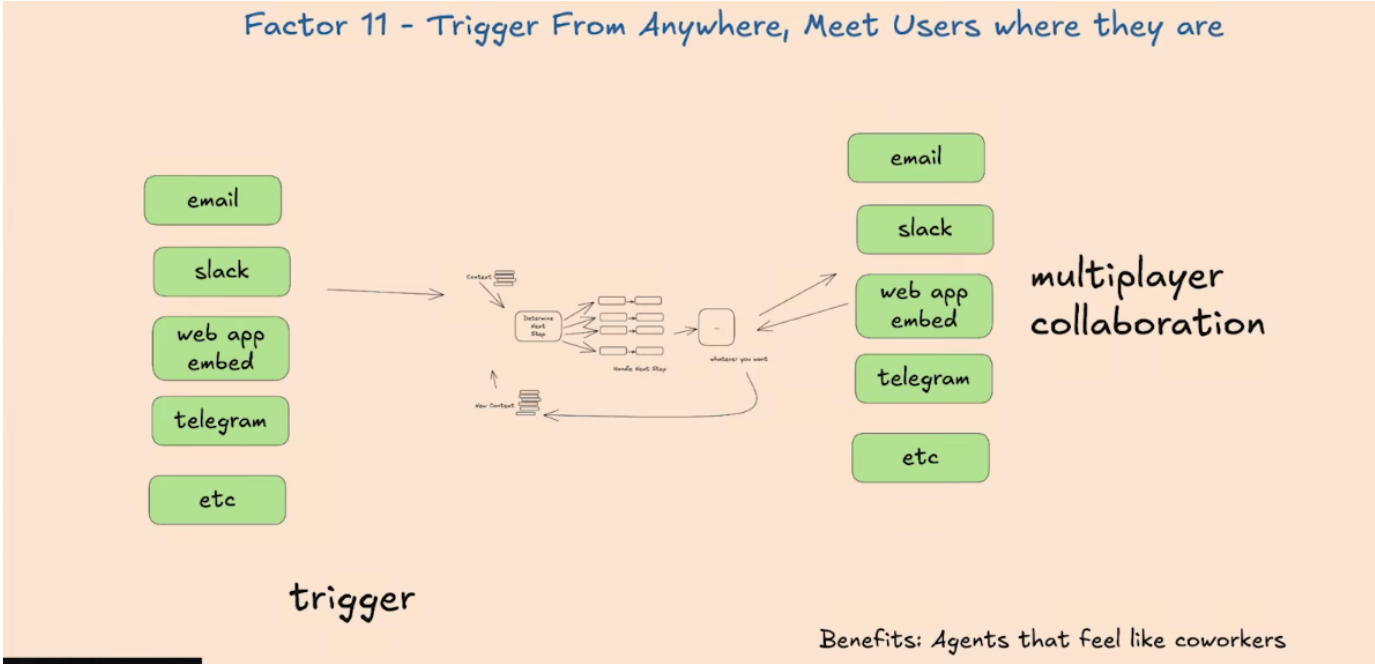
```
tool_calls: [{
  function: "final_answer",
  arguments: {
    message: "the weather in tokyo is 57 and rainy",
  }
}]

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

Final answer vs. I need input vs. call tool is fiddly

Benefits: Clear instructions for the model, inner vs. outer loop, access to multiple humans

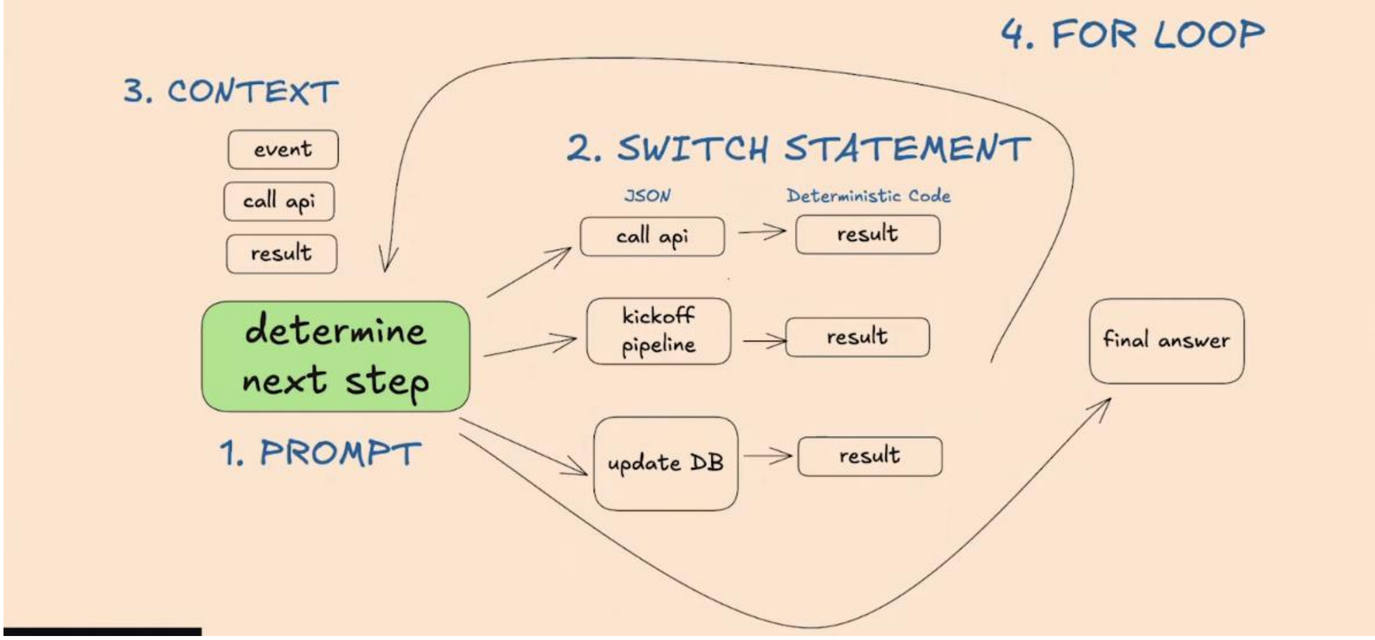
Factor 11
Trigger from anywhere, meet users where they are



Factor 11 - Trigger From Anywhere, Meet Users where they are

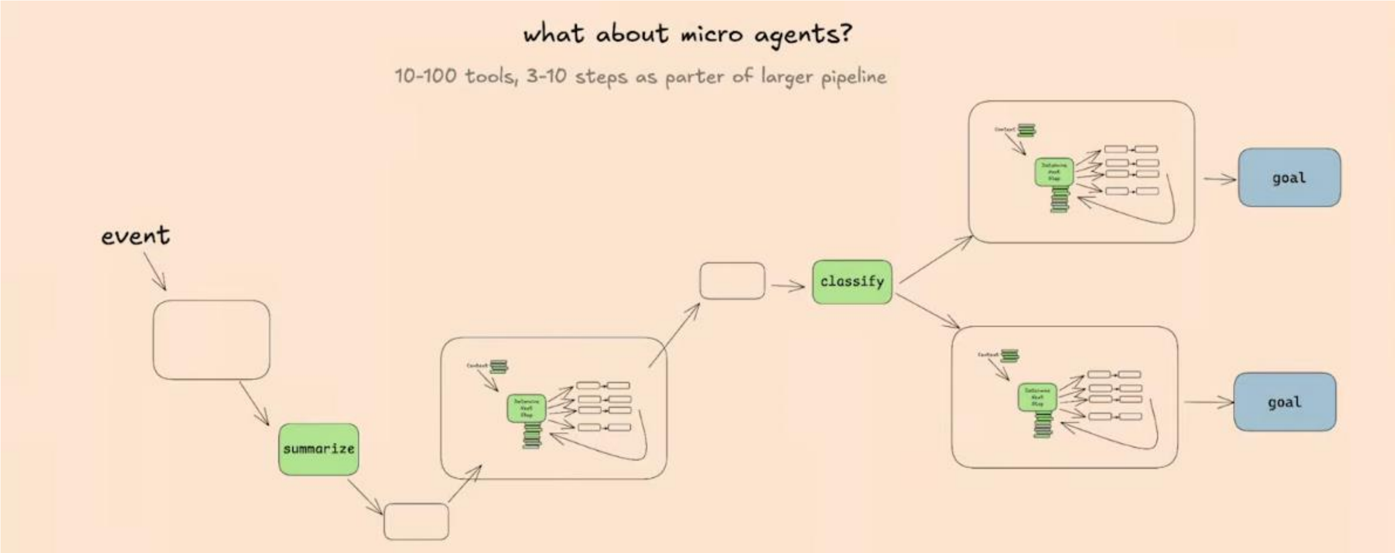
Benefits: Agents that feel like coworkers

Factor 10
Small Focused Agents



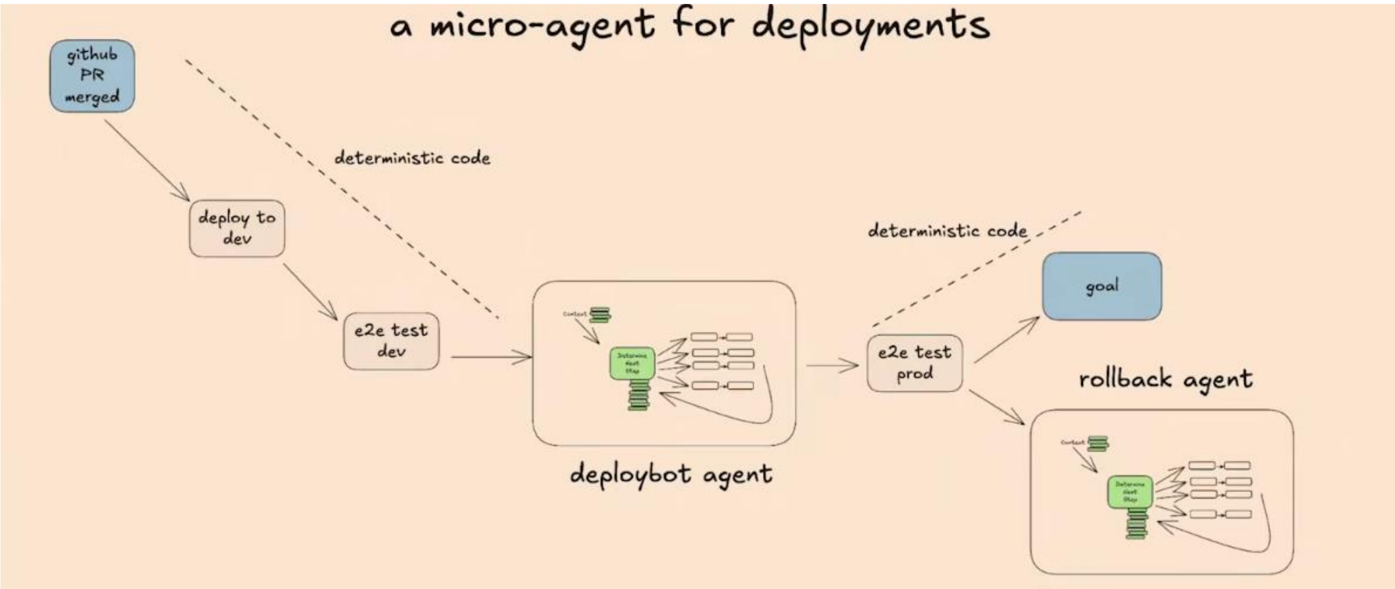
Turns out this doesn’t really work

So what does?

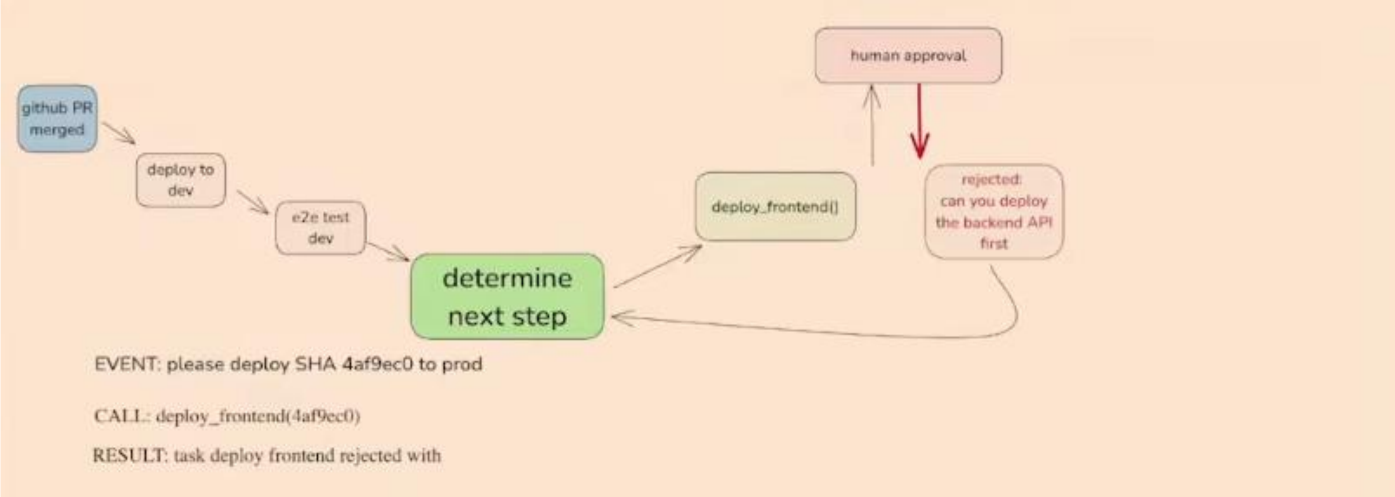


what about micro agents?

10-100 tools, 3-10 steps as parter of larger pipeline



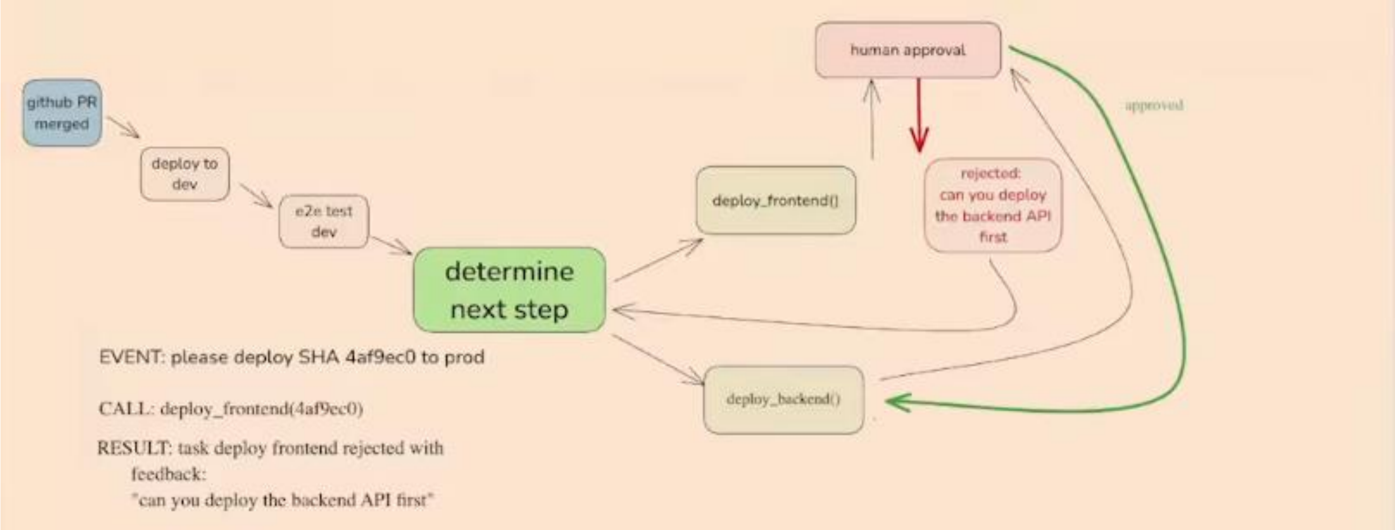
a micro-agent for deployments



EVENT: please deploy SHA 4af9ec0 to prod

CALL: deploy_frontend(4af9ec0)

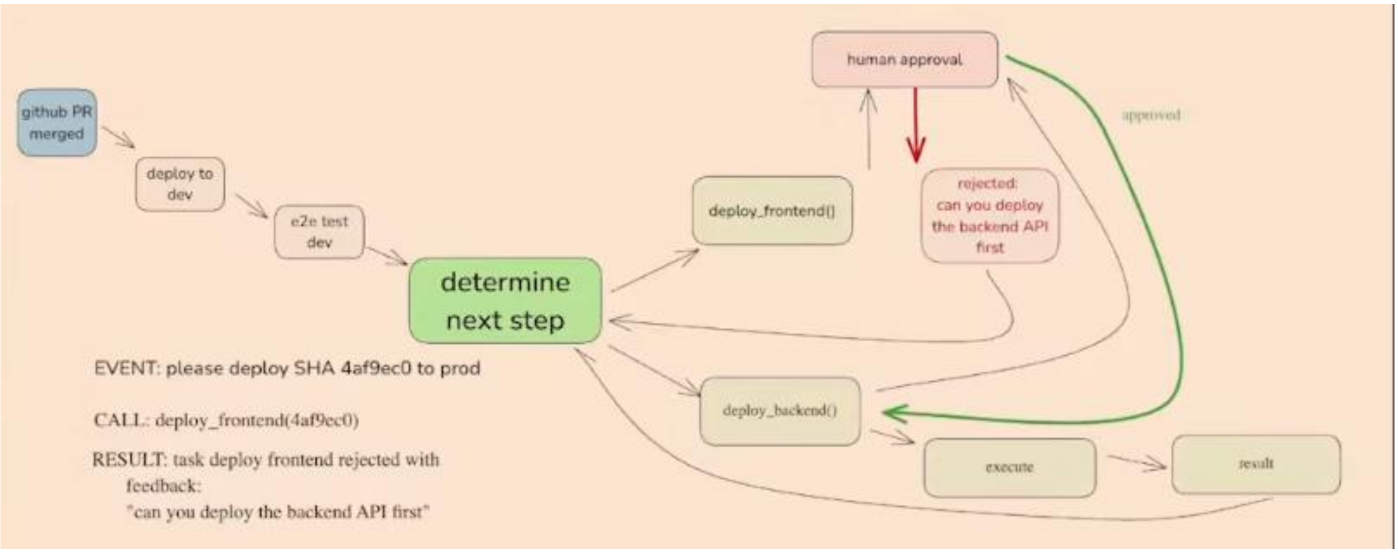
RESULT: task deploy frontend rejected with



EVENT: please deploy SHA 4af9ec0 to prod

CALL: deploy_frontend(4af9ec0)

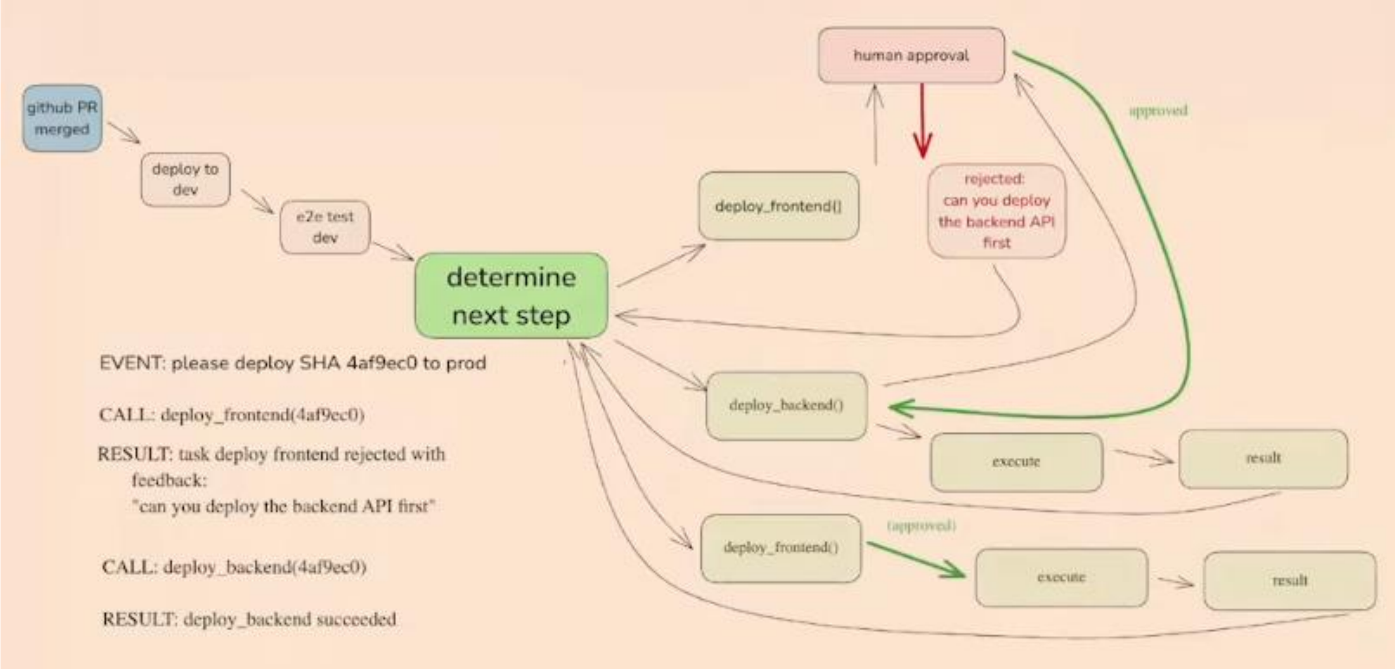
RESULT: task deploy frontend rejected with
feedback:
"can you deploy the backend API first"



EVENT: please deploy SHA 4af9ec0 to prod

CALL: deploy_frontend(4af9ec0)

RESULT: task deploy frontend rejected with
feedback:
"can you deploy the backend API first"



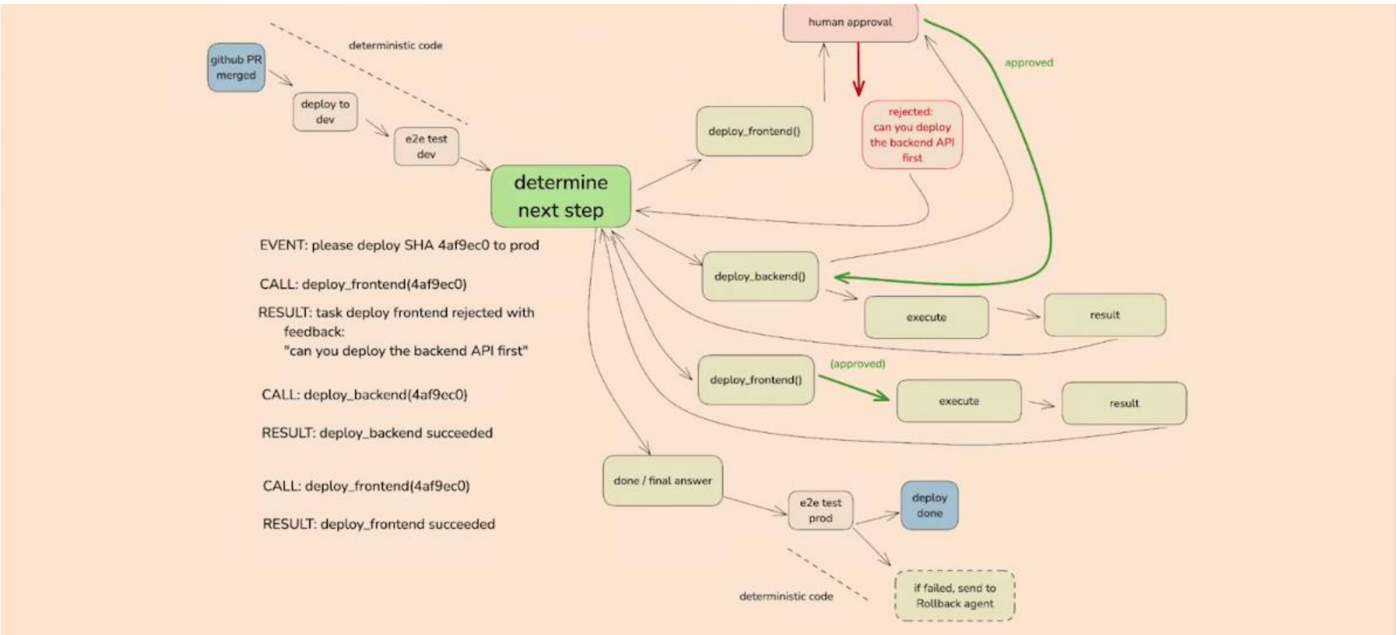
EVENT: please deploy SHA 4af9ec0 to prod

CALL: deploy_frontend(4af9ec0)

RESULT: task deploy frontend rejected with
feedback:
"can you deploy the backend API first"

CALL: deploy_backend(4af9ec0)

RESULT: deploy_backend succeeded



EVENT: please deploy SHA 4af9ec0 to prod

CALL: deploy_frontend(4af9ec0)

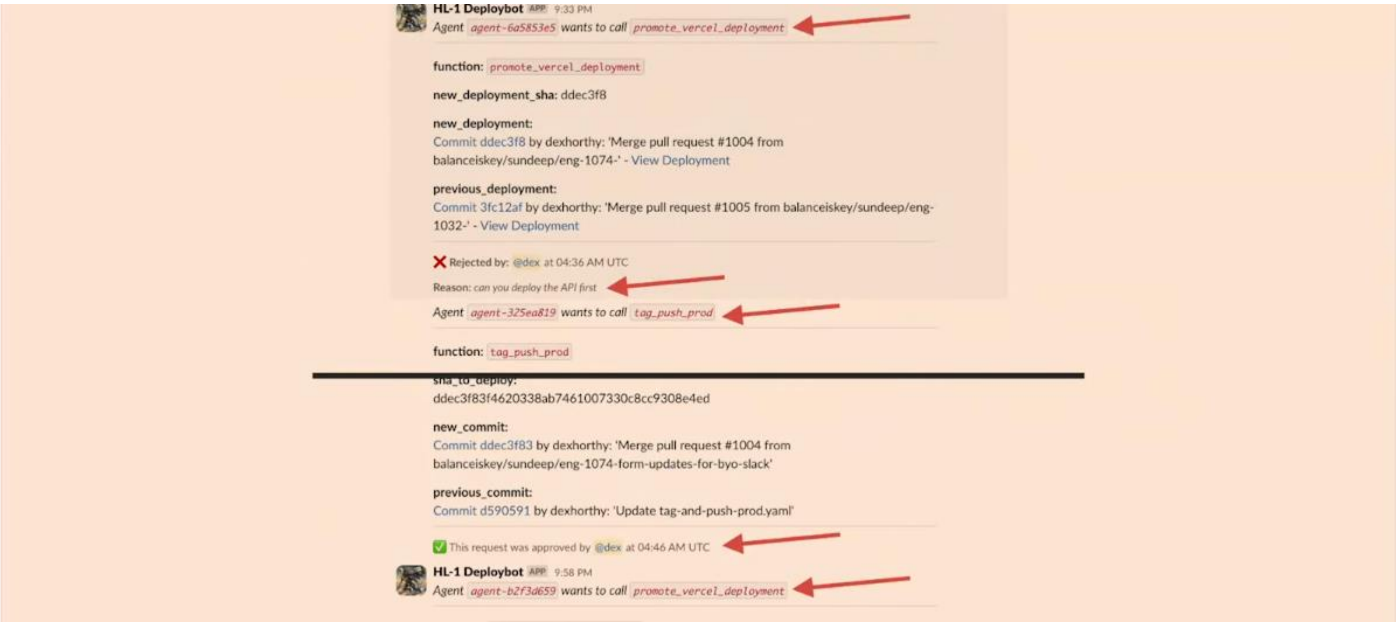
RESULT: task deploy frontend rejected with
feedback:
"can you deploy the backend API first"

CALL: deploy_backend(4af9ec0)

RESULT: deploy_backend succeeded

CALL: deploy_frontend(4af9ec0)

RESULT: deploy_frontend succeeded



HL-1 Deploybot APP 9:33 PM
Agent `agent-6a5853e5` wants to call `promote_vercel_deployment`

function: `promote_vercel_deployment`

new_deployment_sha: ddec3f8

new_deployment:
Commit ddec3f8 by dexhorthy: 'Merge pull request #1004 from balanceiskey/sundeep/eng-1074-' - View Deployment

previous_deployment:
Commit 3fc12af by dexhorthy: 'Merge pull request #1005 from balanceiskey/sundeep/eng-1032-' - View Deployment

❌ Rejected by: @dex at 04:36 AM UTC

Reason: *can you deploy the API first*

Agent `agent-325ea819` wants to call `tag_push_prod`

function: `tag_push_prod`

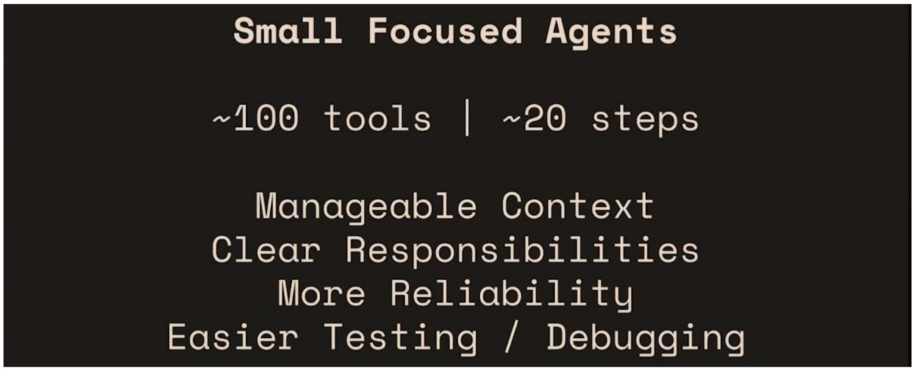
sha_to_deploy:
ddec3f83f4620338ab7461007330c8cc9308e4ed

new_commit:
Commit ddec3f83 by dexhorthy: 'Merge pull request #1004 from balanceiskey/sundeep/eng-1074-form-updates-for-byo-slack'

previous_commit:
Commit d590591 by dexhorthy: 'Update tag-and-push-prod.yaml'

✅ This request was approved by @dex at 04:46 AM UTC

HL-1 Deploybot APP 9:58 PM
Agent `agent-b2f3d659` wants to call `promote_vercel_deployment`

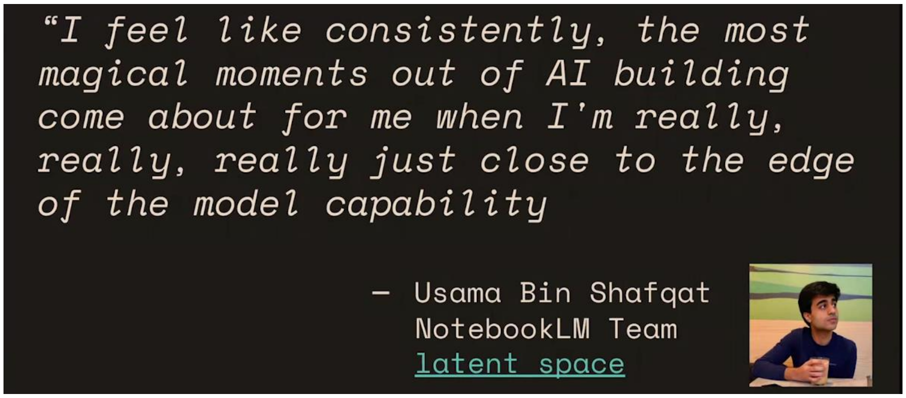


Small Focused Agents

~100 tools | ~20 steps

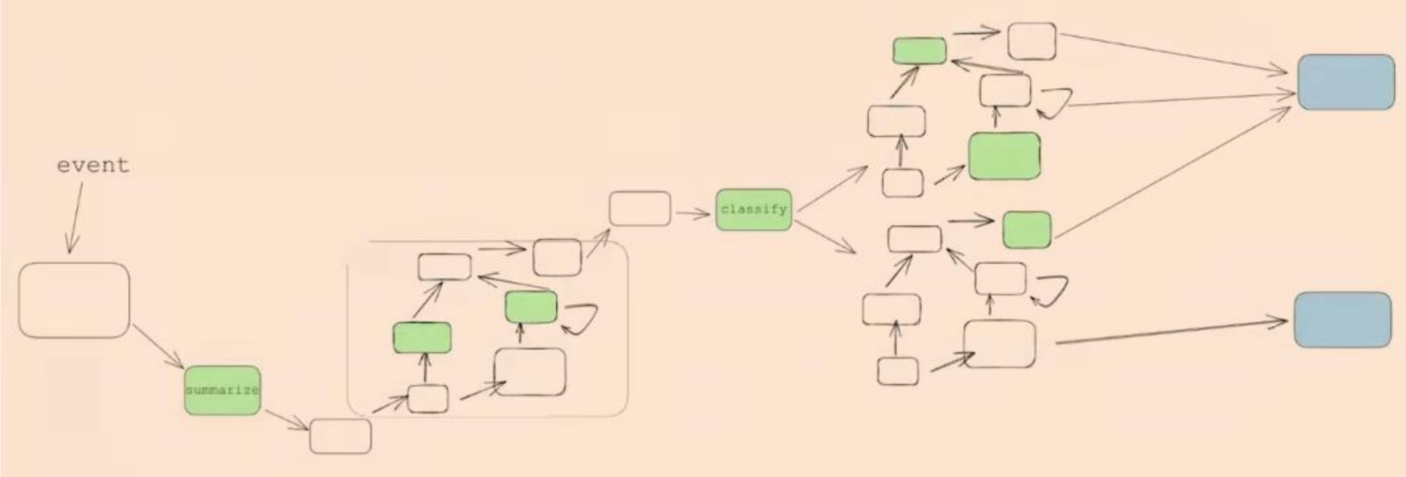
Manageable Context
Clear Responsibilities
More Reliability
Easier Testing / Debugging

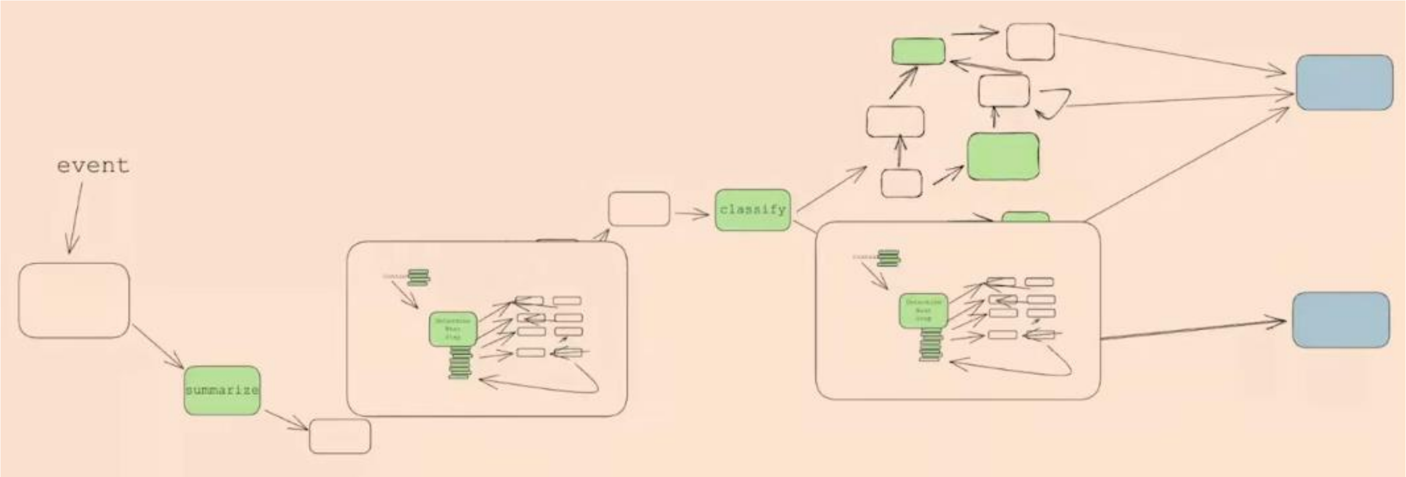
What if LLMs keep getting smarter?

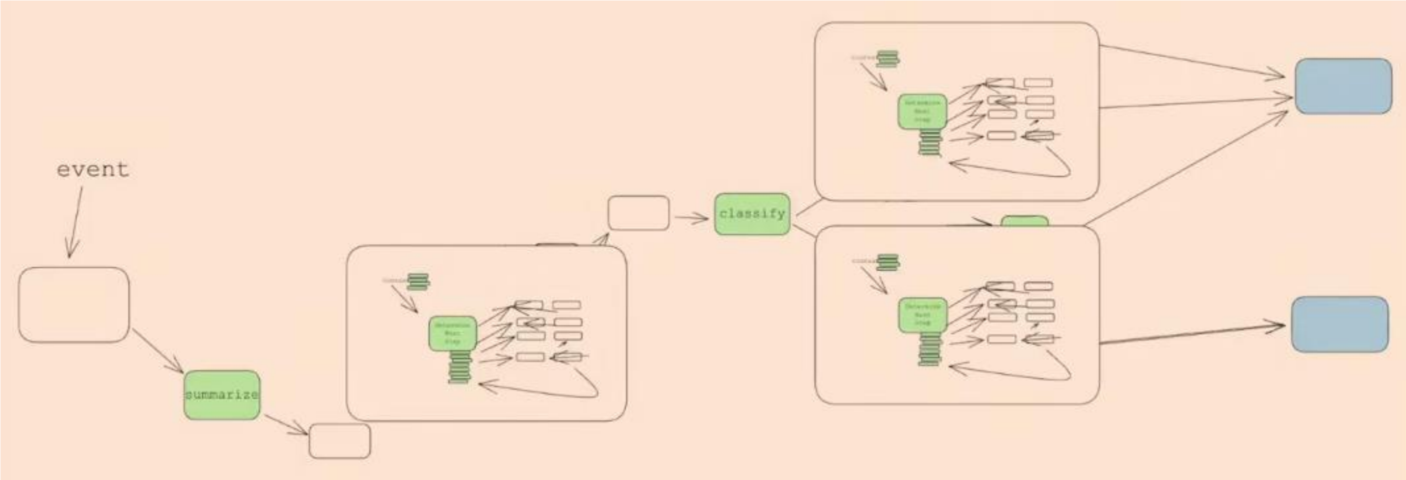


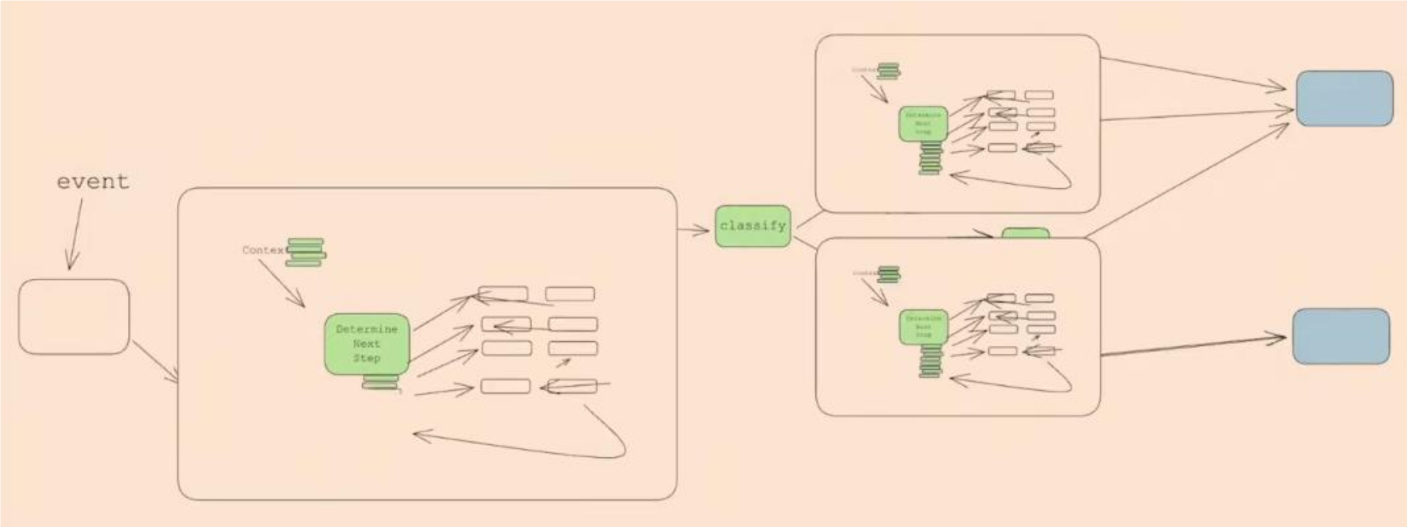
“I feel like consistently, the most magical moments out of AI building come about for me when I’m really, really, really just close to the edge of the model capability

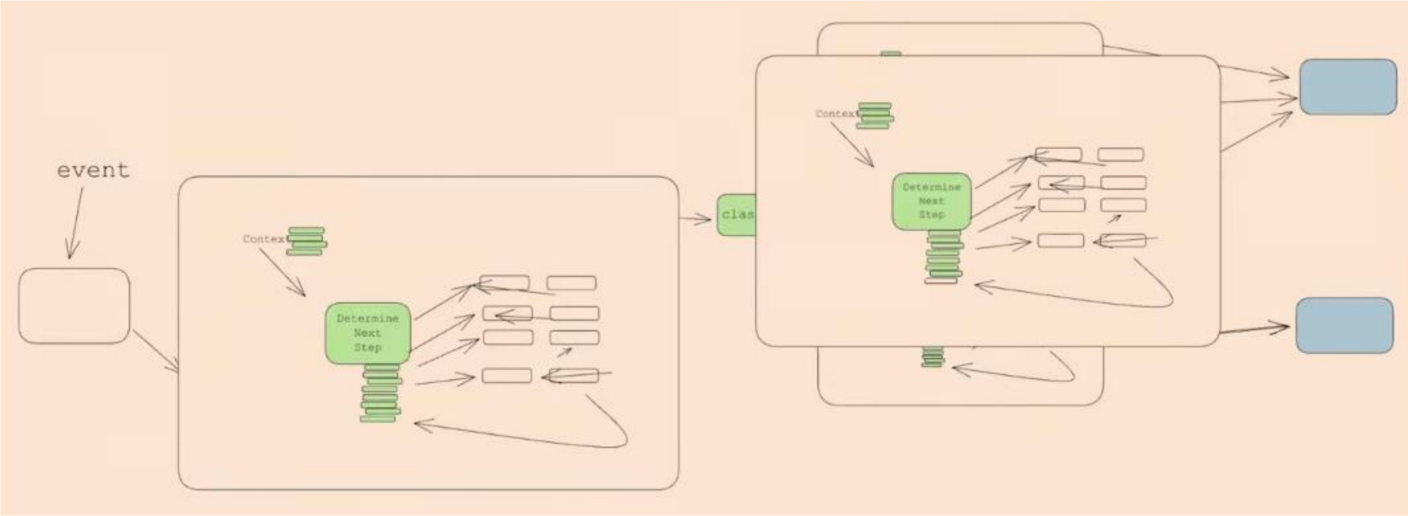
— Usama Bin Shafqat
NotebookLM Team
[latent space]

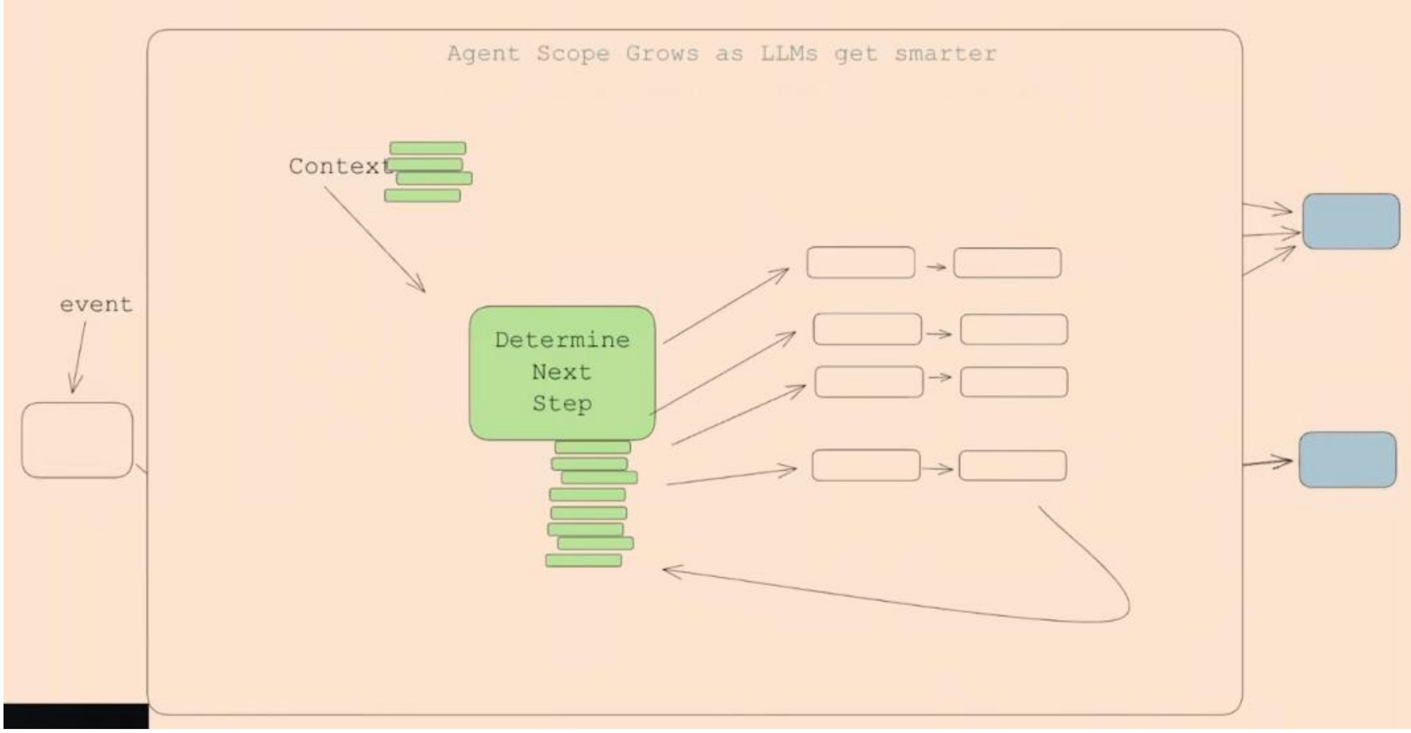


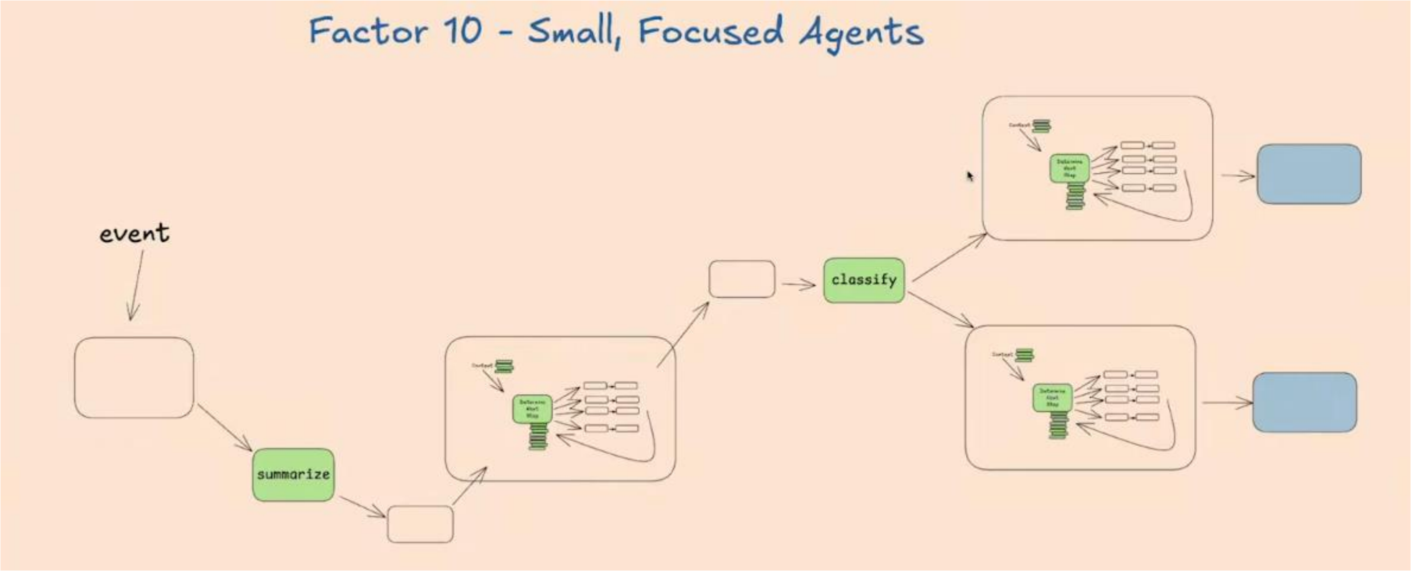








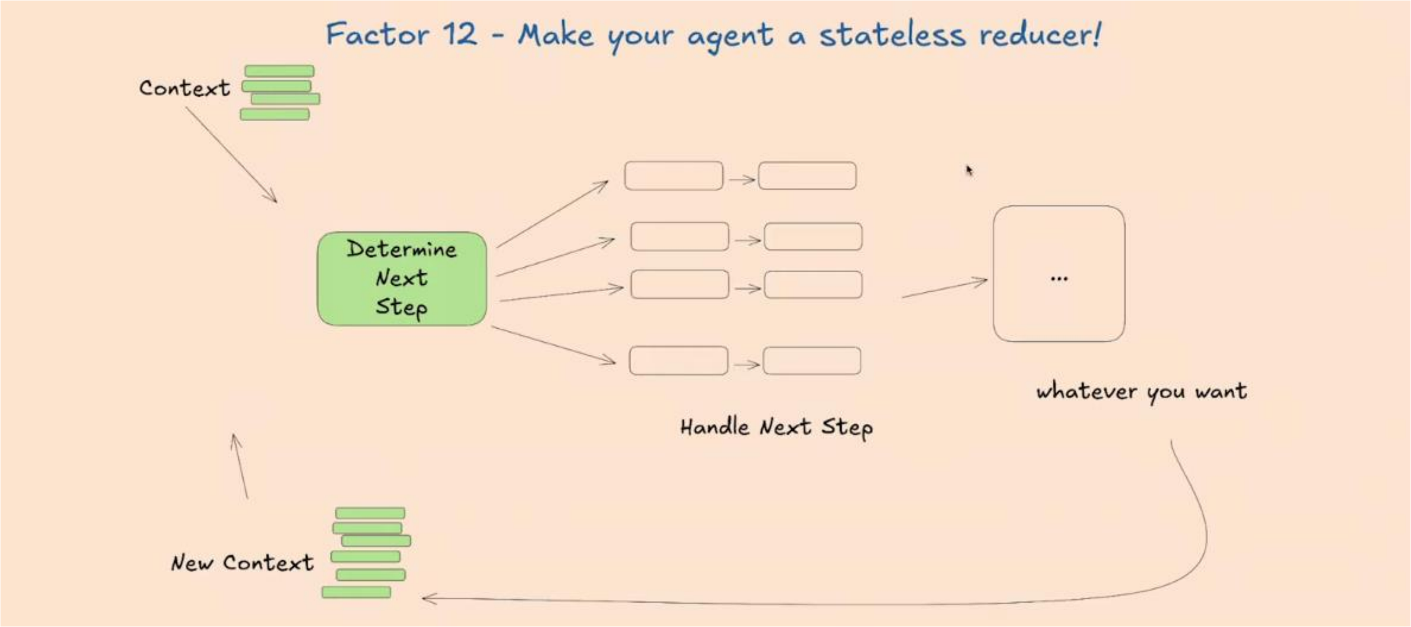

Agent Scope Grows as LLMs get smarter



Factor 10 - Small, Focused Agents

Factor 12

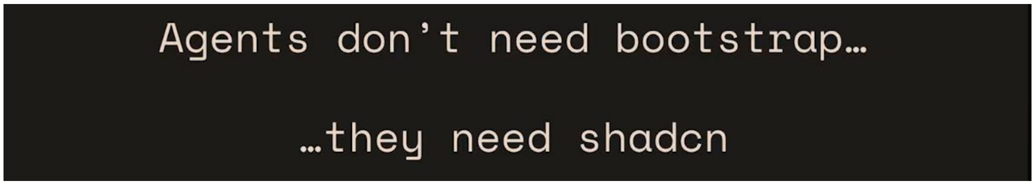
On stateless reducers



Factor 12 - Make your agent a stateless reducer!

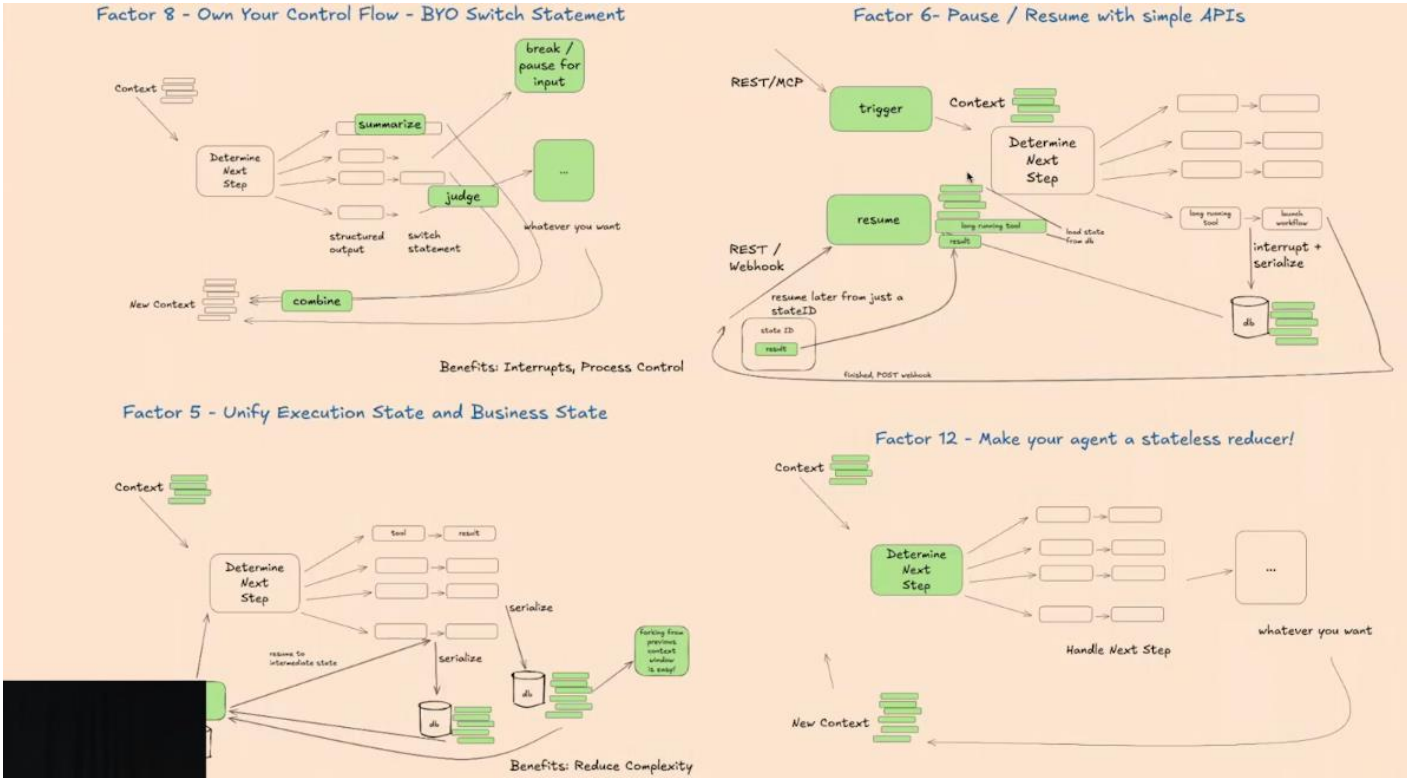
We're still finding the right abstractions

```
$ npx create-12-factor-agent
```



Agents don't need bootstrap…

…they need shadcn



Factor 8 - Own Your Control Flow - BYO Switch Statement

Benefits: Interrupts, Process Control

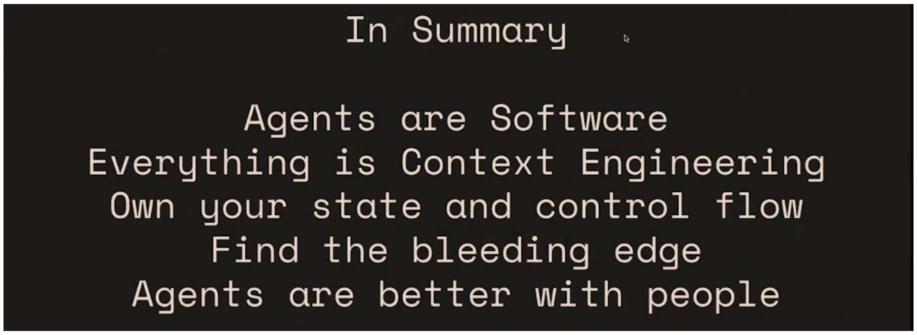
Factor 6- Pause / Resume with simple APIs

Factor 5 - Unify Execution State and Business State

Context
Determine Next Step
tool
result
serialize
serialize
resume to intermediate state
db
db
forking from previous context window is easy!

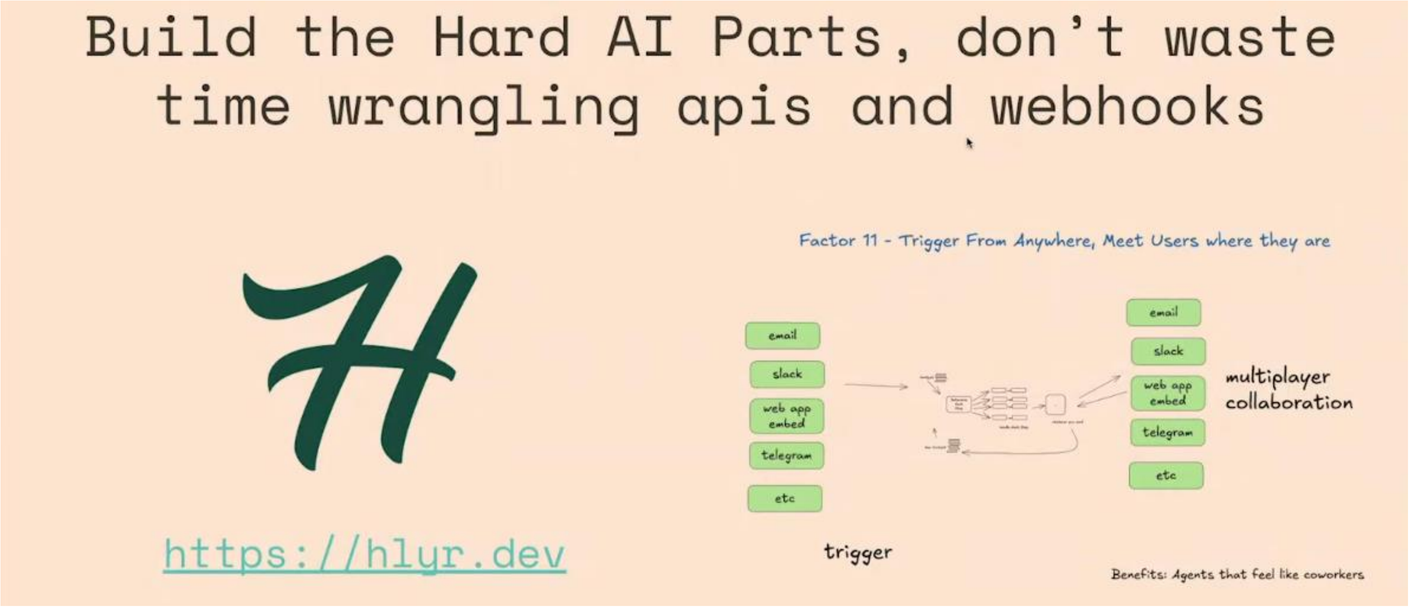
Benefits: Reduce Complexity

Factor 12 - Make your agent a stateless reducer!



In Summary

Agents are Software
Everything is Context Engineering
Own your state and control flow
Find the bleeding edge
Agents are better with people

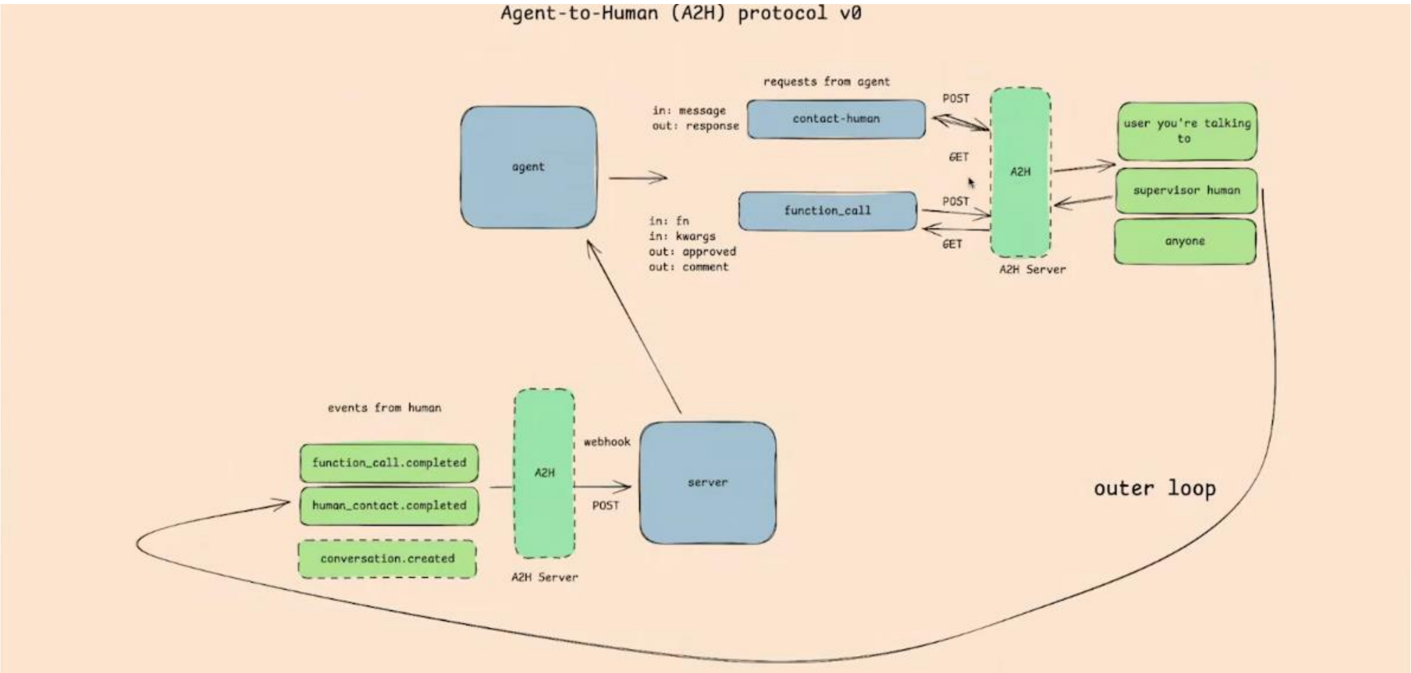


Build the Hard AI Parts, don't waste time wrangling apis and webhooks

https://hlyr.dev

Factor 11 - Trigger From Anywhere, Meet Users where they are

Benefits: Agents that feel like coworkers



Agent-to-Human (A2H) protocol v0