TNSA

A Guide to Build **Agents** with NGen and MCP.

Contents

What are Agents?

- Autonomous orchestrators that plan, call tools/APIs, and iterate until a goal is achieved.
- Core loop:
    1. Think/plan (LLM proposes next action)
    2. Act (call a tool/API)
    3. Observe (ingest results)
    4. Reflect (adjust plan or finish)
- Why now? Tooling like MCP standardizes how agents invoke thousands of real-world actions safely and consistently.

Building Agents

- Inputs: User goal + context (history, docs, state)
- Capabilities: Reasoning, tool-use, memory, safety, and recovery
- Outputs: A final result (answer, file, state change), plus an audit trail
- Patterns:
    - Planner-Executor: One LLM plans and uses tools; simple, reliable
    - Multi-Tool Router: The agent selects the right tool among many
    - RAG-First: Agent augments reasoning with retrieval/context

Understanding MCP and Its Implications

- MCP (Model Context Protocol) provides a uniform interface to tools across 7000+ apps (via Zapier MCP), including auth, schemas, and execution.
- Implications:
    - No per-app SDKs needed; tools are discoverable via list_tools()
    - Stronger security posture: treat the MCP server URL like a secret
    - Scalability: add new tools without redeploying the agent backend
    - Observability: tool executions are auditable

How to use MCP with TNSA

- API endpoints:
    - GET /mcp/tools — lists available tools from your Zapier MCP server
    - POST /mcp/call — executes any tool by name with JSON params
- Python client:
    - Lists tools and calls tools using fastmcp
    - Reads ZAPIER_MCP_SERVER_URL from .env

What makes NGen better for Agents and MCP?

- Hosted, enterprise-grade serving. No local setup required.
- Streaming-ready with optimized connection pooling and concurrency controls.
- Deterministic tool-use scaffolding: prompts tuned for concise, structured tool call intents.
- Cost-aware: tracks tokens and costs, enabling safe loops.

Implementation Samples

These examples demonstrate how to integrate agents and MCP tools without exposing internal architecture. Bring your own MCP server URL via .env (ZAPIER_MCP_SERVER_URL).

1) List tools (cURL)

```
curl -sS -X GET "$TNSA_API_BASE/mcp/tools" -H "Authorization: Bearer
$TNSA_API_KEY"
```

2) Call a tool (cURL)

```
curl -sS -X POST "$TNSA_API_BASE/mcp/call" \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $TNSA_API_KEY" \
  -d
'{"tool_name":"gmail_find_email","params":{"query":"from:billing@example.com
"}}'
```

3) Python: list tools

```
import os,requests
API_BASE = os.environ["TNSA_API_BASE"]
headers = {"Authorization": f"Bearer {os.environ['TNSA_API_KEY']}"}
resp = requests.get(f"{API_BASE}/mcp/tools", headers=headers,timeout=30)
resp.raise_for_status() for t in resp.json().get("tools", []):
print(t.get("name"), "-", t.get("description", ""))
```

4) Python: call a tool

```
import os, requests

API_BASE = os.environ["TNSA_API_BASE"]
payload = {"tool_name": "slack_send_message", "params": {"channel": "#ops",
"text": "Deployed ✅"}}
resp = requests.post(f"{API_BASE}/mcp/call", json=payload,
```

```
headers={"Authorization": f"Bearer {os.environ['TNSA_API_KEY']}"},
timeout=60)
print(resp.json())
```

5) Node.js (fetch): call a tool

```
import fetch from "node-fetch";

const API_BASE = process.env.TNSA_API_BASE;
const res = await fetch(`${API_BASE}/mcp/call`, {
  method: "POST",
  headers: {"Content-Type": "application/json", "Authorization": `Bearer
${process.env.TNSA_API_KEY}`},
  body: JSON.stringify({ tool_name: "notion_create_page", params: { title:
"Q3 Plan" } }),
});
console.log(await res.json());
```

6) Minimal agent loop (Python)

```
import os, requests, json

API_BASE = os.environ["TNSA_API_BASE"]
def model_complete(prompt: str) -> str:
    r = requests.post(f"{API_BASE}/generate", json={"prompt": prompt,
"max_tokens": 300}, timeout=60)
    r.raise_for_status(); return r.json()["text"]

def try_extract_tool_call(text: str):
    try:
        obj = json.loads(text)
        if isinstance(obj, dict) and "tool" in obj and "params" in obj:
            return obj["tool"], obj["params"], obj.get("reason", "")
    except Exception:
        return None

goal = "Summarize unread support tickets and post to Slack #support"
ctx = f"Goal: {goal}\nWhen you need a tool, respond ONLY with JSON
{{\"tool\":...,\"params\":...,\"reason\":...}}."
history = []
for _ in range(5):
    out = model_complete(ctx + "\n" + "\n".join(history))
```

```
    tc = try_extract_tool_call(out)
    if not tc:
        print("FINAL:", out); break
    tool, params, why = tc
    tr = requests.post(f"{API_BASE}/mcp/call", json={"tool_name": tool,
"params": params}, timeout=60)
    tr.raise_for_status()
    history.append(f"TOOL[{tool}] => {tr.text}")
```

7) Agent prompt template (copy-paste)

You are a capable operator. If a tool is needed, output ONLY JSON:

```
{"tool":"<name>","params":{...},"reason":"..."}. Otherwise, answer directly.
```

8) Validation before calling tools (Python)

```
import jsonschema

def validate_params(params, schema):
    jsonschema.validate(instance=params, schema=schema)
```

9) JavaScript: list tools and pick by description

```
const res = await fetch(`${process.env.TNSA_API_BASE}/mcp/tools`, { headers:
{ Authorization: `Bearer ${process.env.TNSA_API_KEY}` } });
const data = await res.json();
const emailTool = data.tools.find(t => /email/i.test(t.description));
```

10) Rate-limit safe retries (Python)

```
import time, requests
for i in range(5):
    r = requests.post(f"{API_BASE}/mcp/call", json=payload)
    if r.status_code != 429: break
    time.sleep(2 * (i+1))
```

11) Streaming UI hint

Use POST /infer for token streaming (SSE) to display incremental responses while tools are running.

12) BYO MCP server

Set your Zapier MCP server URL in .env as ZAPIER_MCP_SERVER_URL. Treat it as a secret. Each team can point to their own MCP server with their connected apps.

13) Pagination pattern for tool results (Python)

```python
def paged_call(tool, params):
    page = 1
    while True:
        p = {**params, "page": page}
        r = requests.post(f"{API_BASE}/mcp/call", json={"tool_name": tool,
"params": p}, timeout=60)
        r.raise_for_status()
        data = r.json()
        yield data
        if not data.get("has_more"): break
        page += 1
```

14) Schema-driven form generation (JS)

```javascript
// Fetch tool schema and render a simple form
const tools = await (await
fetch(`${process.env.TNSA_API_BASE}/mcp/tools`)).json();
const t = tools.tools.find(x => x.name === "calendar_create_event");
const schema = t?.input_schema; // JSON Schema
// Iterate schema.properties to render inputs; respect required[]
```

15) Role-based allowlist check (pseudo)

```
If user.role == "analyst": allow tools:
["gmail_find_email","sheets_append_row"]
If user.role == "ops": add
["slack_send_message","pagerduty_create_incident"]
```

```
Enforce before calling /mcp/call
```

16) Distributed tracing context (HTTP headers)

```
curl -H "X-Trace-Id: $(uuidgen)" -H "X-Span-Id: $(uuidgen)" \
  -H "Authorization: Bearer $TNSA_API_KEY" \
  "$TNSA_API_BASE/mcp/call" -d '{"tool_name":"...","params":{}}'
```

17) Error normalization (Python)

```
def normalize_error(resp):
    try:
        j = resp.json()
        return {"status": resp.status_code, "error": j.get("error") or j}
    except Exception:
        return {"status": resp.status_code, "error": resp.text[:2000]}
```

18) Simple SDK: Agent generate (Python)

```
import os, requests

class TNSAClient:
    def __init__(self, base=None, api_key=None):
        self.base = base or os.environ["TNSA_API_BASE"]
        self.headers = {"Authorization": f"Bearer {api_key or
os.environ['TNSA_API_KEY']}", "Content-Type": "application/json"}

    def agent_generate(self, goal: str, max_steps: int = 5, system: str |
None = None):
        payload = {"goal": goal, "max_steps": max_steps}
        if system: payload["system"] = system
        r = requests.post(f"{self.base}/agent/generate", json=payload,
headers=self.headers, timeout=120)
        r.raise_for_status(); return r.json()
```

19) Simple SDK: Agent generate (JS/TS)

```
export async function agentGenerate(goal: string, maxSteps = 5, system?:
string) {
  const res = await fetch(`${process.env.TNSA_API_BASE}/agent/generate`, {
    method: "POST",
    headers: {"Content-Type": "application/json", Authorization: `Bearer
${process.env.TNSA_API_KEY}`},
    body: JSON.stringify({ goal, max_steps: maxSteps, system }),
  });
  if (!res.ok) throw new Error(`agent/generate failed: ${res.status}`);
  return res.json();
}
```

20) Cache tool list client-side (JS)

```
let toolCache; let cachedAt = 0;
export async function getTools() {
  const ttl = 5 * 60 * 1000;
  if (toolCache && Date.now() - cachedAt < ttl) return toolCache;
  const res = await fetch(`${process.env.TNSA_API_BASE}/mcp/tools`, {
headers: { Authorization: `Bearer ${process.env.TNSA_API_KEY}` }});
  toolCache = await res.json(); cachedAt = Date.now(); return toolCache;
}
```

Conclusion

- With MCP + NGen, you can build robust agents that orchestrate actions across thousands of apps without per-app code.
- This repo provides:
  - Generic MCP integration (/mcp/tools, /mcp/call)
  - NGen generation endpoints (/infer, /generate)
  - A simple Python MCP client (sdk_py/tnsa_mcp/client.py)
- Next steps:
  - Add a small tool-use loop endpoint (e.g., /agent/generate) to automate plan-act-observe cycles.
  - Cache /mcp/tools responses for faster UI.
  - Add role-based allowlists for sensitive tools.