ChatGPT

FountainKit Concept Paper: Swift as the LLM Superpower

The Superpower Pitch

Most LLM-based systems today can *talk about code* or *propose edits*, but very few can actually **instantiate, compile, and run** typed API clients on demand *in their native language*. By enabling an LLM to execute Swift code inside a secure sandbox—and pairing it with Apple's official Swift OpenAPI Generator plugin—we unlock a true superpower. This approach gives the LLM the ability to turn an **abstract API specification** (OpenAPI) directly into a **first-class, type-safe, executable tool** within its own environment, in real time [1]. In other words, the agent can:

- **Read a spec**,
- **Generate a typed Swift client** (via Apple's Swift OpenAPI Generator [2]),
- **Compile and run it securely** in a Docker-based sandbox, and
- **Invoke those APIs deterministically** through the strongly-typed interfaces.

This loop – *spec* → *code* → *execution* – is what makes FountainKit more than just another LLM runtime. It essentially becomes a *self-extending operating system for intelligence*, where the LLM can continuously extend its own capabilities by spinning up new tools from specs on the fly [3].

Why This Matters

1. Spec-Driven Development, in Real-Time

Traditional spec-driven development means humans write client code from an OpenAPI document to ensure correct API usage [4]. FountainKit brings this into real-time: given an API spec, the LLM immediately generates a Swift client for it and uses it. Every external capability is exposed to the agent as a type-safe Swift library, *on demand*. The LLM isn't guessing how to call an API from text or crafting HTTP calls blindly – it's calling *stable, compiled Swift functions* (e.g. `UsersAPI.getProfile(id:)`) that were generated directly from the API's formal spec [1]. This guarantees correctness to the spec and eliminates improvisation.

2. One Language, End-to-End

FountainOS (the runtime environment for the agent) is written in Swift. The agent **reasons in Swift**, orchestrates tools in Swift, and executes Swift code. There's no impedance mismatch between thinking, planning, and doing – they all speak the same language. This unified end-to-end Swift approach simplifies the architecture and makes the agent's chain of thought directly executable. (By contrast, many AI agents juggle multiple languages or formats between reasoning and execution, which introduces translation overhead and potential errors.)

3. Strong Typing as Guardrails

Swift's static type system enforces correctness at compile time, acting as powerful guardrails for the agent [5]. The compiled Swift client comes with all the safety features of the language:

- **Optionals** mean the LLM must handle the case when data is missing, rather than blindly assuming its existence.
- **Enums** constrain values to only the allowed cases (preventing invalid parameters).
- **Codable structs/models** guarantee that JSON responses precisely match the expected schema, or decoding will fail clearly.

In short, Swift is a *type-safe* language that catches many errors during compile time rather than at runtime [5]. These properties serve as guardrails for the LLM's actions – the agent can't, for example, call an undefined endpoint or mis-type a field name without the compiler catching it. The result is higher reliability in tool usage compared to free-form code generation in dynamic languages.

4. Deterministic and Auditable

Every action the agent takes is a **concrete Swift code snippet**. When the agent decides to use a tool or perform some operation, it produces a snippet of Swift code to do so. That code is then compiled and executed, producing results. This process is deterministic and reproducible: given the same snippet and environment, you will get the same result every time. We log each code execution with the source snippet, its output, and its exit status. This means we have a complete audit trail of the agent's "thoughts" and behaviors in code. If something goes wrong or an unexpected result is produced, developers can replay or debug the exact sequence of steps (down to line numbers and output). This level of transparency and determinism is a stark contrast to black-box end-to-end prompt reasoning – here every intermediate step is an auditable artifact (the compiled binary and its logs).

5. Safe yet Limitless

Thanks to a hardened Docker sandbox, the agent operates in a *safe* environment without sacrificing capability. We impose strict runtime restrictions on the code execution container – for example, it runs as a non-root user, with a read-only filesystem, no outbound network access, limited CPU/memory, and no privilege escalation. These constraints mean the agent can execute arbitrary Swift code *as if it had limitless abilities*, but in reality all actions occur within strict guardrails set by the host operator. Security best practices (like dropping all Linux capabilities and using seccomp/AppArmor profiles to filter system calls) ensure that even if the LLM generates malicious or errant code, it cannot escape or cause unintended harm [6] [7]. Industry experts note that the only robust way to run AI-generated code safely is to **isolate** it from the host environment [8] [9] – exactly what our sandbox does. The model feels empowered to use any tool or code it needs, while we remain confident that it's fenced in by the Docker container's policies.

Architecture Overview

```
      +--------------------+
      |   FountainKit      |       (Agent’s Tool Registry – in Swift)
      +--------------------+
                |
      +--------------------+
      |  Swift API Clients |       (Generated from OpenAPI specs)
      |  (spec-driven)     |
      +--------------------+
                 |
      +--------------------+
      |   Swift Sandbox    |       (Docker-hardened execution)
      |  - swift build/run |
      |  - swift test      |
      +--------------------+
```

Figure: ***FountainKit high-level architecture.*** The LLM agent consults a registry of available tools (all implemented in Swift). Many of these tools are *Swift API clients generated from OpenAPI specifications*. When the agent selects a tool, the call is executed inside a secure Swift Sandbox. The sandbox is a Dockerized Swift runtime that can compile and run code safely (including running any Swift package tests if needed). This design allows the set of tools to expand dynamically (the agent can generate new Swift packages/clients from specs and add them to the registry) while keeping execution isolated.

Reference Swift Sandbox Implementation

To concretize the sandbox design, below is a reference **docker-compose service** definition for a hardened Swift sandbox. This configuration is tuned for security and determinism in executing Swift code for the agent:

```
services:
  swift-runner:
    image: swift:5.10
    user: "1000:1000"                # Run as non-root user for safety
    read_only: true                  # Immutable filesystem (except work
dirs)
    tmpfs:
      - /tmp:size=256m,mode=1777     # Ephemeral tmp storage
      - /run:size=32m,mode=0755      # Required for Swift runtime
    volumes:
      - type: bind                   # Writable scratch workspace for code
        source: ./scratch
        target: /workspace
        read_only: false
      - type: bind                   # Pre-populated SwiftPM cache (for deps)
        source: ./spm-cache
        target: /home/swift/.swiftpm
        read_only: true
    cap_drop:
      - ALL                         # Drop all Linux capabilities (secure by
default)
    security_opt:
      - no-new-privileges:true      # No privilege escalation
    pids_limit: 256                 # Limit number of processes
    ulimits:
```

```
    nproc: 512                # Further limit processes/threads
    nofile: 4096              # Limit open files
  deploy:
    resources:
      limits:
        cpus: '2.0'           # CPU quota for the container
        memory: 2g            # Memory limit for the container
  network_mode: "none"        # Disable networking (no outbound access)
```

Hardened Runtime Policies: The above sandbox is configured with multiple layers of defense:

- **No network access:** The container has `network_mode: "none"`, meaning the executed code cannot make external calls (unless explicitly allowed via a different configuration). This ensures all API access is done through the vetted Swift clients. (By default, Docker containers have no access to the host network or filesystem, which keeps generated code contained [10]. We only enable networking if a tool explicitly requires it, and even then it can be tightly controlled.)
- **Read-only filesystem:** Except for a designated `/workspace` scratch volume (for compiling and temporary files), the filesystem inside the container is read-only. The Swift code cannot modify the container's OS or any important files.
- **Minimal Linux capabilities:** We drop **all** Linux capabilities in the container (`cap_drop: - ALL`), and set `no-new-privileges:true` [6]. This means even if the Swift process tried, it cannot acquire elevated privileges or do anything requiring admin rights.
- **Seccomp/AppArmor profiles:** We recommend running the container with a strict seccomp profile (e.g., Docker's default or a custom profile) and AppArmor enforcement. Seccomp filters restrict which system calls the process can make, so dangerous syscalls result in a safe failure (`EPERM`) [7]. This effectively sandboxes the process at the kernel level.
- **Cgroups resource quotas:** The `deploy.resources.limits` ensure the container uses at most 2 CPU cores and 2GB of RAM. The `pids_limit` and `ulimits` cap how many processes or threads can be spawned. These prevent runaway code (e.g. infinite fork bombs or memory leaks) from impacting the host.
- **Timeouts for runaway code:** On top of Docker's limits, the execution engine will enforce wall-clock timeouts for any code run. For example, a Swift snippet that runs longer than *N* seconds will be terminated. This prevents infinite loops from hanging the agent.
- **Audit logs:** Every code execution in the sandbox is logged with details: a hash of the code, the timestamp, resources used, stdout/stderr output, and the exit code. These audit logs make it possible to trace exactly what the agent did and when, which is crucial for debugging and safety review.

Minimal Toolchain Inside the Image: The Docker image for the sandbox is kept minimal yet functional for on-demand code execution. It includes just the essentials:

- The Swift compiler and toolchain (enabling use of `swiftc`, `swift build`, `swift run`, `swift test`).
- Apple's **swift-openapi-generator** plugin and the **swift-openapi-runtime** library (plus an HTTP client transport library like AsyncHTTPClient). These allow the container to generate and run API clients from OpenAPI specs offline.
- A pre-cached Swift Package Manager (SPM) artifacts directory (`~/.swiftpm`), vendored with any dependencies the generated clients might need. This way the container does not need network access to fetch packages at runtime – everything is resolved at build time.

- The FountainKit tool registry and any built-in tools are baked into the image. This means when the container starts, it already knows about the standard tools and any pre-generated API clients (from specs we included at image build time).

End-to-End Flow: Putting it all together, the FountainKit workflow looks like this:

1. **Build time (CI or Image Build):** We pre-generate as much as possible. For each known OpenAPI spec, we run the Swift OpenAPI Generator to produce a Swift client library [2] . These client libraries (and their dependencies) are compiled and stored in the Docker image (or mounted volume). Essentially, we "vend" a suite of tools into the image ahead of time. This step happens in a CI pipeline or Docker build process, not by the LLM.
2. **Runtime (LLM in action):** When a user's request comes in, the LLM (agent) decides which tools to use. If it needs to call an external API for which a client was pre-generated, it can directly invoke that Swift library through FountainKit. If it encounters a *new* spec or needs a novel piece of code, the LLM can also generate a fresh Swift snippet or even a new Swift package on the fly. FountainKit will compile and execute that in the sandbox. The key is that at runtime the agent is orchestrating Swift code execution (either calling existing tool functions or creating new ones) to accomplish the task.
3. **Governance and Oversight:** All actions are logged, and a policy engine can intervene if something looks suspicious. For example, if the agent somehow tries to use `Process` to spawn a shell or make a network call outside allowed channels, those calls would be caught by the sandbox (and our monitoring) and can be stopped. Operators can review the audit logs and have fine-grained control – essentially a kill-switch or approval step for certain privileged operations. This ensures that even as the agent extends itself, it remains under **trusted supervision**.

Conclusion

Allowing the LLM to execute Swift inside a hardened sandbox isn't *just another feature* – it's a transformation of what the LLM can do. FountainKit becomes a self-extending operating system for the agent, where the LLM can:

- **Turn specs into running code** (instantly generating new capabilities from API descriptions),
- **Use robust, typed APIs instead of brittle text-based prompts or HTTP calls**,
- **Operate entirely in its native language (Swift)** for every step of reasoning and execution,
- All while staying **within safe, deterministic boundaries** defined by the sandbox.

This is the "superpower" we set out to achieve – and it's a true differentiator against the usual Codex-style systems that only *talk about* code. We've given the LLM a real compiler and a real runtime, with real-world APIs at its fingertips, safely [11] . The result is an AI agent that feels far more capable and grounded: it can literally **write an app for itself** to solve a task, use it, and do so with the assurances of type safety and sandbox security.

Next Steps: Going forward, we can cement this concept with a practical prototype. We propose preparing a minimal **demo repository** – for example, a Dockerfile and a lightweight service (e.g. a FastAPI or Vapor server) that accepts Swift code from the LLM, compiles and runs it in the sandbox, and returns the output logs. This would allow us to immediately test the full loop ("LLM submits Swift snippet → compile & run in container → return results") in a controlled setting. Such a demo would make the ideas in this concept paper concrete and runnable, proving out the FountainKit approach end-to-end and providing a foundation for further iteration.

---

[1] [2] [4] Introducing Swift OpenAPI Generator | Swift.org
https://www.swift.org/blog/introducing-swift-openapi-generator/

[3] [9] [10] Build Your Own Code Interpreter - Dynamic Tool Generation and Execution With o3-mini
https://cookbook.openai.com/examples/object_oriented_agentic_approach/secure_code_interpreter_tool_for_llm_agents

[5] Why Choose Swift for Your Next Mobile App? Key Insights for Large ...
https://emphasoft.com/blog/why-choose-swift-for-your-next-mobile-app/

[6] Secure AI Agents at Runtime with Docker
https://www.docker.com/blog/secure-ai-agents-runtime-security/

[7] Running Untrusted Python Code — Andrew Healey
https://healeycodes.com/running-untrusted-python-code

[8] Secure Code Execution in AI Agents | by Saurabh Shukla | Medium
https://saurabh-shukla.medium.com/secure-code-execution-in-ai-agents-d2ad84cbec97

[11] The Code Interpreter by OpenAI: Redefining the Analysis Programming Landscape
https://promptengineering.org/the-code-interpreter-by-openai-redefining-the-analysis-programming-landscape/