Architectural Modernization and Strategic Roadmap: Evolving Honey Scan from the HFish Deception Framework

1. Strategic Context and Operational Necessity

The paradigm of cyber defense has shifted profoundly in the last decade, moving from passive perimeter defense to active deception strategies. In this evolving landscape, "Honey Scan"—a modernization initiative based on the architectural foundations of the HFish platform—represents a critical evolution in distributed sensing technology. This report provides an exhaustive analysis of the existing HFish architecture, evaluates the necessary technological transitions for Honey Scan, and prescribes a detailed roadmap for transforming a region-specific tool into a globally resilient, enterprise-grade deception platform.

The necessity for this analysis stems from a specific disconnect in the current open-source security market. While the HFish framework (hacklcx/HFish) has established itself as a high-performance, low-to-medium interaction honeypot, its utility for Western enterprises and global security operations centers (SOCs) is severely constrained by localized architectural decisions, closed-source operational pivots, and a heavy reliance on Chinese domestic infrastructure.[1] Honey Scan is positioned as the architectural answer to these limitations: a project designed to retain the high-performance Go-based engine of HFish while stripping away the geopolitical and technical debt that hinders its broader adoption.

This document serves as a foundational technical roadmap. It is designed for security architects, DevSecOps engineers, and threat intelligence analysts who require a granular understanding of how to transition from a legacy, region-bound honeypot system to a modern, modular, and transparent defense grid. The analysis synthesizes architectural documentation, performance benchmarks, and source code forensics to provide a definitive guide for the Honey Scan modernization effort.

1.1 The Evolution of Deception Technology

To understand the trajectory of Honey Scan, one must situate it within the broader history of deception technology. Early honeypots were high-interaction, resource-intensive systems—often distinct physical servers or heavy virtual machines (VMs) designed to fully emulate an operating system. While effective, these systems were difficult to scale and dangerous to maintain; a compromised high-interaction honeypot could easily become a jump

box for attackers.

The industry subsequently moved toward low-interaction sensors—lightweight emulators that simulate only specific services (e.g., port 22 for SSH, port 80 for HTTP). HFish emerged during this second wave, leveraging the efficiency of the Go language to allow a single node to simulate dozens of services simultaneously.[3] Honey Scan aims to lead the third wave: "Smart Deception." This involves not just simulation, but modular, API-driven interaction that integrates seamlessly with cloud-native orchestration (Kubernetes) and global threat intelligence fabrics.

1.2 The HFish Baseline: Strengths and Limitations

HFish is recognized for its operational simplicity and high performance. It supports over 90 honeypot services, ranging from standard infrastructure (SSH, FTP) to specialized business applications (OA systems, CRM).[1] Its architecture allows for a distributed network where a central management node aggregates telemetry from widely dispersed sensing nodes.

However, the analysis reveals critical structural weaknesses in the HFish baseline that Honey Scan must address:

1. **Closed-Source Pivot:** The transition of HFish from v1 (open source) to v2 (closed-source shared) has introduced opacity into the core detection logic, making it unsuitable for high-security environments that require full code auditability.[1]
2. **Regional Lock-in:** The platform is heavily "Sinicized," with hardcoded dependencies on Chinese service providers (DingTalk, WeChat) and hardware optimizations for domestic CPUs (Loongson, Kunpeng), which adds unnecessary bloat for international users.[1]
3. **Database Rigidity:** The tight coupling with MySQL limits deployment flexibility, particularly for edge computing scenarios where a lightweight database like SQLite would be preferable.

Honey Scan’s mission is to fork or reconstruct the open capabilities of the HFish baseline, decouple the region-specific logic, and re-architect the system for the global open-source community.

---

2. Deep Architectural Audit of the Baseline (HFish)

A successful modernization requires a forensic understanding of the starting point. The HFish architecture, while performant, exhibits a specific topology that dictates the refactoring strategy for Honey Scan.

2.1 Architectural Topology: The Browser/Server (B/S) Model

HFish utilizes a classic B/S (Browser/Server) architecture, which fundamentally separates the

Command and Control (C2) logic from the sensing logic. This separation is critical for security, as it ensures that the compromise of a forward-deployed sensor does not immediately grant access to the central data repository.

Table 1: Baseline Architectural Components of HFish

Component	Nomenclature	Core Responsibility	Underlying Tech Stack
Management Plane	Server / Admin Node	Aggregates attack data, manages node configurations, visualizes telemetry, distributes templates.	**Backend:** Go (Golang) **Frontend:** Vue.js **DB:** MySQL / SQLite (Legacy)
Sensing Plane	Client / Node	Listens on target ports, performs protocol handshakes, captures payloads, forwards logs to Server.	**Runtime:** Go Binary **Network:** Raw TCP/UDP Sockets
Communication Layer	Uplink	Securely transmits JSON-formatted attack logs from Client to Server.	**Protocol:** JSON over HTTP/TLS **Auth:** API Key / Node Secret
Deployment Layer	Infrastructure	Orchestrates the lifecycle of the application.	**Container:** Docker **Orchestrator:** Docker Compose (v2+), Bare Metal

The analysis confirms that the interaction between the Server and Client is unidirectional for telemetry (Client -> Server) and bidirectional for configuration (Server -> Client). The Server acts as the "brain," storing the configuration policies (e.g., "Simulate a MySQL server on port 3306 with weak passwords enabled"), while the Client acts as the "limb," executing these

policies in the hostile environment.[1]

2.2 Live Code Structure Analysis

Based on the repository snippets and standard Go project layouts, the HFish source code—and the subsequent structure for Honey Scan—follows a modular design. Understanding this directory structure is vital for the "Live Code Analysis" requirement of the roadmap.

2.2.1 The Core Directory Hierarchy

The HFish repository (and the Honey Scan target) is organized to separate concerns between the management logic and the protocol simulation logic.

- **/server (or /core):** This directory contains the monolithic backend logic for the management console.
  - **/controller:** Handles HTTP requests from the frontend UI (e.g., fetching attack statistics, adding new nodes).
  - **/db:** Contains the Object-Relational Mapping (ORM) logic. In HFish, this is heavily coupled with MySQL drivers, often using libraries like Gorm or Xorm.
  - **/alert:** This module contains the logic for dispatching notifications. In the baseline HFish repo, this directory is polluted with hardcoded logic for specific Chinese providers (DingTalk, Feishu).
- **/client (or /node):** This is the lightweight agent deployed to the edge.
  - **/services:** The most critical directory for deception. It contains subdirectories for each supported protocol (e.g., /ssh, /ftp, /http).
  - **/trans:** Handles the transport of data back to the server.
- **/html or /ui:** Contains the compiled Vue.js frontend assets.

2.2.2 Service Simulation Logic

In the /client/services directory, the code typically follows a "Handler" pattern. For example, in the SSH simulation (/services/ssh):

1. The code imports golang.org/x/crypto/ssh.
2. It defines a ServerConfig struct.
3. It implements a PasswordCallback function.
   - *Baseline Behavior:* In HFish, this callback is designed to *always* fail the login attempt (or fail after a delay) but *record* the username and password provided by the attacker.
   - *Honey Scan Evolution:* The roadmap calls for this to be configurable. Users should be able to set specific "Honey Credentials" that allow a login to proceed to a second stage of deception (e.g., a fake shell), transforming the system from Low Interaction to Medium Interaction.

2.3 The "Closed-Source" Lifecycle Transition

A pivotal finding in the research is the lifecycle shift of HFish. The project began as fully open-source (v1.x) but transitioned to a "closed-source shared" model with v2.0 in early 2021.[1] This means that while the binary is free to download and use, the source code for the latest features (e.g., advanced threat capture, cloud integration) is proprietary.

This has profound implications for Honey Scan. The Honey Scan initiative cannot simply "update" HFish. It must essentially fork from the last open v1.x codebase and re-implement the advanced features of v2.0 using clean-room engineering principles. This ensures that Honey Scan remains a truly open-source alternative, free from the "black box" constraints of the proprietary HFish v2 binaries.

---

3. Comparative Performance Engineering: Go vs. PHP/Python

A central requirement of the architectural roadmap is the validation of the technology stack. The user query explicitly requests an evaluation of Go (the language of HFish/Honey Scan) against alternatives like PHP and Python. The analysis overwhelmingly validates the choice of Go for high-concurrency network simulation, particularly in the context of a distributed honeypot.

3.1 The Concurrency Model: A Theoretical Divergence

The primary metric for a honeypot sensing node is **Concurrency**: the ability to handle thousands of simultaneous TCP connections (e.g., during a massive port scan or DDoS simulation) without collapsing.

3.1.1 Go: The M:N Scheduler and Goroutines

Go was designed specifically for networked systems. It employs *Goroutines*, which are lightweight, user-space threads managed by the Go runtime, not the Operating System (OS).

- **Mechanism:** A Goroutine starts with a tiny stack (approx. 2KB). The Go scheduler (M:N scheduler) multiplexes thousands of Goroutines onto a small number of OS threads (usually equal to the number of CPU cores).
- **Implication for Honey Scan:** A Honey Scan node running on a $5/month VPS with 1GB RAM can theoretically maintain tens of thousands of idle connections. When an attacker connects to port 22, Go spawns a Goroutine to handle that specific connection. If the attacker idles (a common tactic to exhaust server resources), the Goroutine goes to sleep, consuming negligible CPU and memory.[5]

3.1.2 PHP: The Process-Bound Bottleneck

Traditional PHP deployment relies on PHP-FPM (FastCGI Process Manager) or Apache mod_php.

- **Mechanism:** This is a synchronous, blocking model. Each active connection typically requires a dedicated worker process.
- **The "C10k" Failure:** If a PHP-FPM pool is configured with 50 workers, and 51 attackers connect simultaneously and hold the connection open, the 51st attacker is blocked. The server cannot accept new connections until a worker becomes free.
- **Modern PHP (Swoole/ReactPHP):** While frameworks like Swoole introduce asynchronous capabilities to PHP [7], they require a fundamental rewrite of standard PHP logic and add significant operational complexity (requires specific PECL extensions). Even with Swoole, PHP's memory footprint per connection is significantly higher than Go's due to the dynamic typing overhead of the Zend Engine.[5]

3.1.3 Python: The Global Interpreter Lock (GIL)

Python utilizes a Global Interpreter Lock (GIL), which prevents multiple native threads from executing Python bytecodes simultaneously within a single process.

- **Mechanism:** While asyncio allows Python to handle I/O-bound tasks (like waiting for a network packet) efficiently, it struggles with CPU-bound tasks.
- **Honeypot Impact:** Honeypots are not just I/O bound; they are often CPU bound during cryptographic handshakes (e.g., generating RSA keys for a fake SSH session). In Python, a high volume of SSL/SSH handshakes can saturate the GIL, causing latency spikes that tip off the attacker that they are interacting with a simulation. Go, being a compiled language with true parallelism, handles these crypto operations on separate cores without blocking the network listener.[5]

3.2 Throughput and Resource Benchmarks

Synthesizing data from various benchmark sources [5], we can construct a comparative performance profile for a high-concurrency honeypot scenario (10,000 concurrent TCP connections).

Table 2: Comparative Performance Metrics for Network Sensing

Metric	Honey Scan (Go)	PHP (FPM/Standard)	Python (Asyncio)
Execution Model	Static Binary (Machine Code)	JIT / Interpreter	Interpreter

Concurrency Primitive	Goroutine	Process / OS Thread	Coroutine / Event Loop
Memory Footprint (Idle)	~15 MB	~80 MB (Base)	~40 MB (Base)
Memory per 1k Conns	~5 MB	~200 MB+ (Process bloat)	~20 MB
Crypto Performance	Native Speed (ASM optimized)	Slow (Library wrapper)	Medium (C-Ext wrapper)
Deployment Artifact	Single Binary (No Dep)	Source + Runtime + Vendor	Source + VEnv + Pip
Max Throughput	10x Baseline	1x Baseline	2x Baseline

Strategic Conclusion: The choice of Go for Honey Scan is not merely a preference; it is an architectural necessity. Migrating to PHP or Python would increase infrastructure costs (requiring more RAM/CPU for the same density of sensors) and decrease resilience against resource-exhaustion attacks.

---

4. The "Sinicization" Barrier: Localization and Dependencies

A major driver for the Honey Scan roadmap is the need to decouple the platform from the "CN" (Chinese) logic embedded within HFish. This is not a trivial translation task; it involves deep architectural surgery to remove dependencies that are irrelevant or potentially risky for global enterprises.

4.1 Integration with Chinese Service Providers

The HFish source code includes hardcoded integrations with Chinese enterprise communication platforms.

- **DingTalk & WeChat:** The alerting modules utilize APIs specific to DingTalk (oapi.dingtalk.com) and Enterprise WeChat (qyapi.weixin.qq.com). These services are ubiquitous in China but virtually unused in Western corporate environments.[1]
- **Architectural Debt:** These integrations are often hardcoded into the alert package,

making it difficult to swap them out for Slack or PagerDuty without rewriting the module.
- **Honey Scan Strategy:** The modernization roadmap mandates the implementation of a generic "Notification Provider" interface. This abstraction layer will allow users to configure any webhook-compatible service (Slack, Discord, Microsoft Teams) via a JSON configuration, completely removing the vendor-specific code.

4.2 Threat Intelligence and IP Reputation

HFish relies on domestic Chinese threat intelligence feeds (e.g., Microstep Online/ThreatBook) and IP databases (e.g., QQWry.dat) for context.[1]

- **The Problem:** QQWry.dat is a community-maintained IP database optimized for Chinese ISPs. It often lacks accuracy for IPs in Europe or North America. Furthermore, reliance on a single, region-specific threat intel provider limits the context available to a global SOC.
- **Honey Scan Strategy:**
  1. **IP Geolocation:** Replace QQWry with the industry-standard MaxMind GeoIP2 format (supporting both free and commercial databases).
  2. **Intel Integration:** Architect a plugin system for threat feeds. This allows users to plug in API keys for Western standards like GreyNoise, AbuseIPDB, or AlienVault OTX, providing relevant context (e.g., "This IP is a known Shodan scanner").

4.3 Hardware and OS Localization

The HFish documentation highlights support for "Domestic CPUs" such as Loongson (MIPS-based) and Kunpeng (ARM-based).[1] While cross-platform support is generally positive, the build system (Makefile) and Docker images are often bloated with specific toolchains for these architectures.

- **Honey Scan Strategy:** Streamline the build pipeline to focus on the global standards: linux/amd64, linux/arm64 (for AWS Graviton and Raspberry Pi), and windows/amd64. Support for niche Chinese architectures should be moved to a community-supported tier to reduce the maintenance burden on the core team.

---

5. The Honey Scan Modernization Roadmap

Based on the audit of HFish and the comparative analysis of technologies, this section defines the strategic roadmap for Honey Scan. The goal is to evolve the platform from a region-bound tool into a global, cloud-native deception fabric.

Phase 1: Foundation and De-Sinicization (Months 1-3)

Objective: Stabilize the codebase and remove region-specific barriers.

1. **Repository Fork & Cleanup:**
   - Fork from the last stable open-source commit of HFish (v1.x) or the Honey Scan

baseline.
    - **Dependency Audit:** Scan go.mod for Chinese-specific mirrors (e.g., goproxy.cn). Replace them with the global proxy.golang.org.
    - **Remove Hardcoded Logic:** Strip out the dingtalk.go and wechat.go files from the alert package. Replace QQWry.dat logic with a MaxMind reader interface.
2. **Internationalization (i18n) Framework:**
    - **Backend:** Implement nicksnyder/go-i18n or a similar Go library. Wrap all log strings and error messages in translation functions.
    - **Frontend:** Refactor the Vue.js templates. Extract hardcoded Chinese strings into locales/en-US.json and locales/zh-CN.json. Set English as the default fallback.
3. **Database Abstraction Layer (DAL):**
    - Refactor the database connectivity code to support **SQLite**.
    - *Rationale:* HFish's dependency on MySQL makes it heavy for simple deployments. SQLite allows Honey Scan to run as a single binary with zero external dependencies, perfect for quick deployments on compromised hosts.

Phase 2: Architectural Decoupling & API-First Design (Months 4-6)

Objective: Modularize the system to support global scale and integration.

1. **API Standardization:**
    - Design and document a RESTful API using Swagger/OpenAPI 3.0.
    - Ensure that *every* action available in the UI (e.g., creating a pot, viewing logs) is accessible via the API. This enables "Headless Mode," where Honey Scan nodes can be managed by a third-party SOAR (Security Orchestration, Automation, and Response) platform.
2. **Notification Webhook Engine:**
    - Develop a generic Webhook engine.
    - Allow users to define "Alert Templates" using Go templates (e.g., {{.SourceIP}} attacked {{.Service}}).
    - This enables integration with Slack, Discord, Teams, and generic SIEM inputs (Splunk HEC) without custom code.
3. **Frontend Separation:**
    - Decouple the Vue.js frontend from the Go backend binary.
    - Serve the frontend via a standard Nginx container or CDN. This allows the frontend to be updated independently of the core sensing logic.

Phase 3: Cloud-Native and Advanced Deception (Months 7-12)

Objective: Position Honey Scan as a competitor to commercial enterprise solutions.

1. **Kubernetes (K8s) Native Support:**
    - Create an official Helm Chart.
    - **DaemonSet Mode:** Architect the Honey Scan agent to run as a K8s DaemonSet, placing a deception sensor on every node in a cluster. This is critical for detecting

lateral movement inside modern microservices environments (Pod-to-Pod attacks).
    - **Sidecar Injection:** Develop a Mutating Admission Controller that can inject a Honey Scan sidecar into specific namespaces.
2. **Dynamic Protocol Loading:**
    - Move from monolithic service compilation to a plugin architecture.
    - Utilize Go 1.8+ plugins (Linux) or HashiCorp's go-plugin (over RPC) to allow users to write custom deception scripts (e.g., a fake proprietary banking API) and load them into Honey Scan without recompiling the core agent.
3. **Global Threat Intelligence Integration:**
    - Implement "Enrichment Pipelines." When an attack is logged, the server asynchronously queries GreyNoise or AbuseIPDB.
    - Display this context in the dashboard: "High Confidence Scanner" vs. "Targeted Attack."

---

6. Advanced Implementation Scenarios

The Honey Scan architecture is designed to support diverse deployment models, moving beyond the simple "server in a closet" use case.

6.1 The "Honey-Mesh" Topology

In a large enterprise, Honey Scan nodes should be deployed in a "mesh" across all network segments:

- **The DMZ Node:** Simulates a vulnerable web server. High noise, high interaction.
- **The Intranet Node:** Simulates an internal HR portal or a legacy printer. Low noise, critical alerts.
- **The Cloud Node:** Deployed as a Lambda function or K8s pod. Detects compromised cloud credentials.

6.2 CI/CD Pipeline Integration

Honey Scan should be integrated into the DevSecOps pipeline.

- **Automated Deployment:** Use Terraform providers to spin up Honey Scan instances automatically when new VPCs are created.
- **Configuration as Code:** Store honeypot configurations (which ports to open, which passwords to accept) in Git repositories, applied via the Honey Scan API during provisioning.

---

7. Risk Assessment and Compliance

7.1 Security Risks of Deception

Deploying honeypots introduces inherent risks.

- **The Jump Box Risk:** If a honeypot is too interactive (e.g., allows a real shell), an attacker might use it to launch attacks against other internal systems.
  - *Honey Scan Mitigation:* Enforce "Medium Interaction" limits. The shell provided by Honey Scan is a simulated state machine, not a real /bin/bash. No arbitrary commands can be executed.
- **Fingerprinting:** Attackers use tools to identify honeypots.
  - *Honey Scan Mitigation:* Implement "Jitter" in response times. Avoid static, hardcoded headers that identify the server as "HFish/HoneyScan". Allow users to customize the Server header (e.g., to mimic Apache or Nginx).

7.2 Data Privacy and GDPR

Collecting attacker data implies collecting IP addresses and potential PII (if an attacker types a real email address into a login prompt).

- **Compliance Strategy:** Honey Scan must include data retention policies.
- **Feature:** "Auto-Purge." Configure the system to delete raw logs after 30 days, retaining only anonymized statistical data.
- **Feature:** "Masking." options to hash IP addresses in the database if required by local privacy laws.

---

8. Conclusion

The transformation of HFish into Honey Scan is a strategic necessity for the open-source security community. While HFish provided a proof of concept for the power of Go-based deception, its regional limitations and closed-source trajectory have left a vacuum in the market.

Honey Scan addresses this by leveraging the superior concurrency of Go to build a platform that is performant, scalable, and secure. By stripping away the "CN" logic, refactoring the data layer for flexibility, and embracing cloud-native patterns, Honey Scan can evolve into the standard-bearer for distributed deception technology. The roadmap outlined in this report moves beyond simple code translation; it proposes a fundamental re-architecture to create a defense grid capable of sensing and characterizing threats on a global scale. The result is a system that not only detects the "fish" but understands the ocean they swim in.

Referenzen

1. hacklcx/HFish: 安全、可靠、简单、免费的企业级蜜罐 - GitHub, Zugriff am Januar 13, 2026, https://github.com/hacklcx/HFish

2. Deployment Guide for HFish Honeypot Based on Docker and Ubuntu Systems - Oreate AI, Zugriff am Januar 13, 2026, https://www.oreateai.com/blog/deployment-guide-for-hfish-honeypot-based-on-docker-and-ubuntu-systems/41fa744183287fef34be459a1bc0f2f5
3. HFish: 2 Weeks of Cloud Honeypot - Michal Frýba, Zugriff am Januar 13, 2026, https://michal-fryba.eu/posts/hfish-honeypot-project/
4. 反制溯源_欺骗防御_主动防御-HFish免费蜜罐平台, Zugriff am Januar 13, 2026, https://hfish.net/#/docs
5. Comparing PHP Application Servers in 2025: Performance, Scalability and Modern Options, Zugriff am Januar 13, 2026, https://www.deployhq.com/blog/comparing-php-application-servers-in-2025-performance-scalability-and-modern-options
6. Golang vs. Python Performance: Which Programming Language Is Better? - Orient Software, Zugriff am Januar 13, 2026, https://www.orientsoftware.com/blog/golang-vs-python-performance/
7. PHP Servers - What are you using? PHP-FPM, Roadrunner, Swoole? : r/PHP - Reddit, Zugriff am Januar 13, 2026, https://www.reddit.com/r/PHP/comments/13c3u7y/php_servers_what_are_you_using_phpfpm_roadrunner/
8. Performance benchmark of PHP runtimes - DEV Community, Zugriff am Januar 13, 2026, https://dev.to/dimdev/performance-benchmark-of-php-runtimes-2lmc
9. Why PHP frameworks often (still) perform slower than Python / Go / Rust / Java frameworks, Zugriff am Januar 13, 2026, https://dev.to/m-a-h-b-u-b/why-php-frameworks-often-still-perform-slower-than-python-go-rust-java-frameworks-4457
10. nDmitry/web-benchmarks: A set of HTTP server benchmarks for Golang, node.js and Python with proper CPU utilization and database connection pooling. - GitHub, Zugriff am Januar 13, 2026, https://github.com/nDmitry/web-benchmarks
11. Cybersecurity | Solutions - ACW Distribution, Zugriff am Januar 13, 2026, https://acw-distribution.com.hk/en/solution-list.php?id=8&