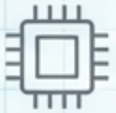
MGraph-AI Service GitHub: Extending API Capabilities

Implementation Blueprint v0.7.0

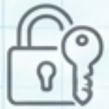
Project: MGraph-AI Service GitHub
Objective: Expose GitHub's Secrets, Actions, and Repos APIs.
Architecture: Stateless, encrypted-PAT proxy.
Target Audience: LLM agents and expert developers.
Reference Codebase: `the-cyber-boardroom/MGraph-AI__Service__GitHub` (branch: `dev`)

Our Mission: Securely Exposing GitHub's Full Power

Current State
Implemented Capabilities



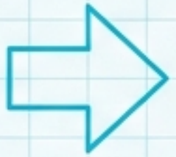
Service
A stateless microservice that proxies GitHub API operations, holding no secrets server-side.



Authentication
Clients provide encrypted GitHub PATs, decrypted on-the-fly with ephemeral keys.

Existing Services
- `GitHub__API` (base operations)
- `GitHub__Secrets` (service layer)
- `Service__Encryption`

Existing Routes
- `/auth/*`
- `/encryption/*`



The Goal
To Be Implemented

New Routes
Expose `GitHub__Secrets`, `GitHub__Actions`, and `GitHub__Repos` services via new, secure REST endpoints.

New Services
Build out the service layer for GitHub Actions (Workflows, Runs, Artifacts) and Repos.

System-Wide Patterns
Solidify the unified request/response schema and shared PAT injection dependency.

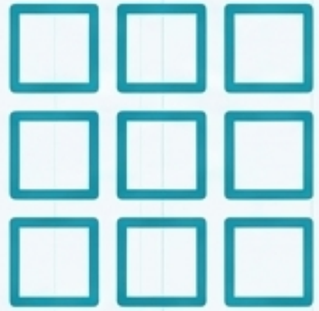
The Three Pillars of Our Architecture

Pillar I: Zero-Trust Security

A completely stateless architecture. The service holds no secrets; credentials are provided per-request and exist only in memory.

Pillar II: Absolute Consistency

A uniform, predictable approach to every request, response, and operation, maximising developer experience and minimising cognitive load.

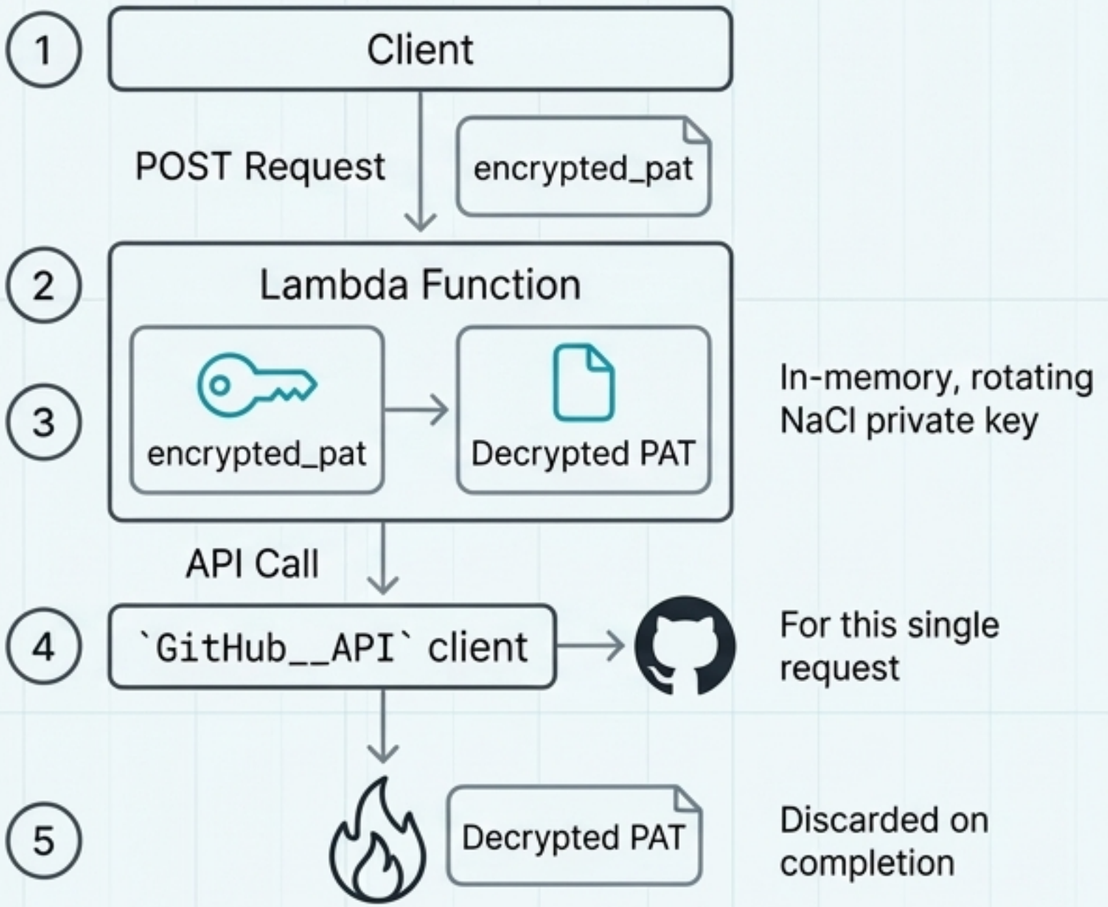


Pillar III: Inherent Safety & Structure

A system built from the ground up with self-validating, type-safe components to ensure robustness and prevent entire classes of errors.

Pillar I: Zero-Trust Security in Practice

The Encrypted PAT Belongs in the Request Body



Rationale

- **Enables Consistency**: Allows for a standard `Schema__GitHub__Request__Base` pattern for all operations.
- **Eliminates GET**: `GET` requests cannot reliably carry request bodies, forcing a consistent `POST`-based approach.
- **Safe for Logging**: The encrypted PAT is an opaque token, meaningless without the private key. It is safe to log, include in error reports, and store in request history.
- **Ephemeral Keys**: The private decryption key is never persisted to disk, exists only in Lambda environment variables, and is rotated on every deployment.

🔗 Pillar II: Driving Consistency Across the API

1. POST for All Read Operations

Explanation: To maintain a consistent request body structure (containing the `encrypted_pat`), we use `POST` for `list` and `get` operations. REST purity is secondary to consistency and security for this internal proxy service.

Operation Type	HTTP Method
List resources	`POST`
Get single resource	`POST`
Create resource	`POST`
Update resource	`PUT`
Delete resource	`DELETE`

2. A Unified Response Schema

Explanation: Every response from the service adheres to the `Schema__GitHub__Response` structure.

```
# Schema__GitHub__Response
class Schema__GitHub__Response(Type_Safe):
    response_context : Schema__GitHub__Response__Context
    response_data    : Schema__Response__Data
```

→ Execution metadata (duration, errors, rate limits)

→ Operation-specific payload (contains partial results on failure)

Pillar III: Building with Inherent Safety & Structure

Everything is `Type_Safe`. All inputs and outputs use `Type_Safe` classes with `Safe_*` primitives.

Runtime Validation: Automatically validates data types at runtime, preventing invalid data from propagating through the system.

Injection Prevention: `Safe_Str` primitives are designed to mitigate injection attacks by default.

Self-Documenting Code: The schemas themselves become the single source of truth for data structures, clear and readable.

Automatic OpenAPI Generation: FastAPI leverages these schemas to generate accurate and interactive API documentation.

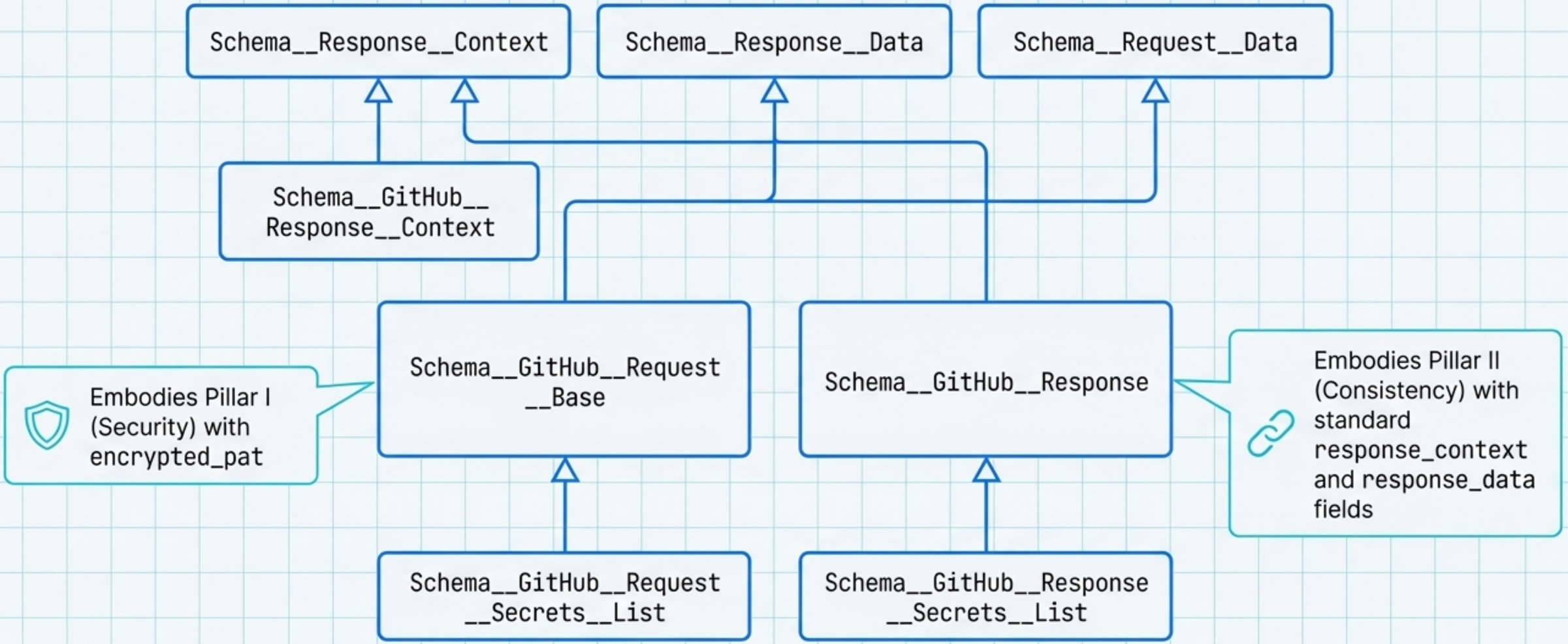
```
# Example Safe Primitives for GitHub

# GitHub username/org (alphanumeric, hyphens, max 39 chars)
Safe_Str__GitHub__Owner

Safe_Str__GitHub__Repo

# Secret name (alphanumeric, underscores)
Safe_Str__GitHub__Secret__Name
```

The Blueprint: Schema Architecture



The Blueprint: Service & Route Hierarchy

mgraph_ai_service_github/service/github/	mgraph_ai_service_github/fast_api/routes/
GitHub__API [Existing]	→ Routes__Auth, Routes__Encryption, Routes__Info [Existing]
GitHub__Secrets [Existing]	→ Routes__GitHub__Secrets__Repo, ...Org, ...Env [New]
GitHub__Actions__Workflows [New]	→ Routes__GitHub__Actions__Workflows, ...Runs, ...Artifacts [New]
GitHub__Actions__Runs [New]	
GitHub__Actions__Artifacts [New]	
GitHub__Repos [New]	→ Routes__GitHub__Repos [New]

[Existing]
[New]

The Central Pattern for Every Route

Every route method follows a standard structure for initialisation, execution, error handling, and response formulation.

① Initialise response objects with default failure state.

② Get configured GitHub_API instance via shared dependency.

⑤ Handle specific, known errors (e.g., bad PAT).

```
def operation_name(self, request: Schema, response: Response) -> Schema:
    with capture_duration() as duration:
        # ①
        response_context = Schema__GitHub__Response__Context(success=False, ...)
        response_data    = Schema__GitHub__Data__*()
        try: # ②
            github_api = self.github_api_factory.get_api(request.encrypted_pat)

            # ③
            result = github_api.get(...)
            response_data.some_field = result.get('field')

            # ④
            response_context.rate_limit = github_api.get_rate_limit()
            response_context.success     = True
            response_context.status_code = Enum__HTTP__Status.OK_200
        except DecryptionError as e: # ⑤
            ...
        except GitHubAPIError as e:  # ⑥
            ...
        finally:
            # ⑦
            response.status_code = response_context.status_code.value
            return Schema__GitHub__Response__*(...)
```

③ Execute operation, populating response_data incrementally.

④ Capture rate limit info and mark success.

⑦ Finalise response and ensure status code is set.

⑥ Map GitHub API errors to our standard error types.

Scope of Work: GitHub Secrets Endpoints

Exposing repository, organisation, and environment secrets management.

/github/secrets/repo/*		
Method	**Path Suffix**	**Operation**
POST	/list	List secrets
POST	/get	Get secret metadata
POST	/create	Create secret
PUT	/update	Update secret
DELETE	/delete	Delete secret

/github/secrets/org/*		
Method	**Path Suffix**	**Operation**
POST	/list	List secrets
POST	/get	Get secret metadata
POST	/create	Create secret
PUT	/update	Update secret
DELETE	/delete	Delete secret

/github/secrets/env/*		
Method	**Path Suffix**	**Operation**
POST	/list	List secrets
POST	/get	Get secret metadata
POST	/create	Create secret
PUT	/update	Update secret
DELETE	/delete	Delete secret

Scope of Work: GitHub Actions Endpoints

Providing control over workflows, runs, and artifacts.

/github/actions/workflows/*		
Method	**Path Suffix**	**Operation**
POST	/list	List workflows
POST	/get	Get workflow details
POST	/dispatch	Trigger workflow run

/github/actions/runs/*		
Method	**Path Suffix**	**Operation**
POST	/list	List workflow runs
POST	/get	Get run details
POST	/cancel	Cancel run

/github/actions/artifacts/*		
Method	**Path Suffix**	**Operation**
POST	/list	List artifacts
POST	/download	Get download URL
DELETE	/delete	Delete artifact

Scope of Work: GitHub Repos Endpoints

Enabling access to repository metadata, branches, and releases.

**Comprehensive Table (`/github/repos/*`)		
Method	**Path**	**Operation**
POST	/get	Get repository info
POST	/branches/list	List branches
POST	/branches/get	Get branch details
POST	/tags/list	List tags
POST	/releases/list	List releases
POST	/releases/get	Get release details
POST	/releases/create	Create release

The Execution Plan: A Phased Approach

1. Foundation (Priority)

- **Tasks:** Implement all Base and GitHub Base schemas, **Enum** definitions, and the shared **GitHub__API__From__Header** dependency.

2. Secrets

- **Tasks:** Implement all Secrets schemas and routes (Repo, Org, Env). Uses the existing **GitHub__Secrets** service.

3. Actions

- **Tasks:** Implement the new service classes (**GitHub__Actions__***), followed by their corresponding schemas and routes.

4. Repos

- **Tasks:** Implement the new **GitHub__Repos** service class, schemas, and routes.

5. Integration

- **Tasks:** Register all new routes in **Service__Fast__API.py** and conduct end-to-end testing.

The Execution Plan: A Three-Layer Testing Strategy

Unit Tests

Focus: Test each service class in isolation.

Method: Mock the `GitHub__API` to verify correct payload construction, response parsing, and business logic for each service method (e.g., `test_GitHub__Actions__Workflows`). (Code snippets in JetBrains Mono).

Route Tests

Focus: Test the FastAPI route layer.

Method: Test request schema validation, response structure, and error handling. Verify that invalid PATs return `401 UNAUTHORIZED`, and that operations failing mid-way return partial results. (Code snippets in JetBrains Mono).

Integration Tests

Focus: End-to-end validation.

Method: Use real (test) GitHub PATs to verify the full lifecycle of operations (e.g., Create -> Get -> Update -> Delete a secret) and confirm `rate_limit` info is correctly populated. (Code snippets in JetBrains Mono).

Definition of Done: Our Success Criteria

- [ ] All schemas defined with `Type_Safe` classes.
- [ ] All new routes implemented and registered, returning correct response schemas.
- [ ] HTTP status codes correctly match operation outcomes.
- [ ] Rate limit info is populated on all GitHub API responses.
- [ ] Encrypted PAT is required in the request body for all GitHub operations.
- [ ] Encrypted secret values are required for create/update secret operations.
- [ ] Unit tests pass for all new service classes.
- [ ] Route and Integration tests pass with a test GitHub PAT.
- [ ] OpenAPI documentation is correctly auto-generated from the new schemas and routes.