Comprehensive Documentation on OpenFaaS with a Focus on Python Functions

I. Introduction to OpenFaaS

A. What is OpenFaaS? (Serverless Functions Made Simple)

OpenFaaS stands as an open-source serverless platform designed to streamline the deployment of event-driven functions and microservices, primarily targeting Kubernetes environments. It significantly reduces the need for repetitive, boilerplate coding, enabling developers to package their code or existing binaries into Docker images. These images then become highly scalable endpoints, equipped with built-in auto-scaling capabilities and comprehensive metrics collection.[1] The platform facilitates rapid development and deployment, adeptly managing spikes in traffic by scaling up resources and efficiently scaling down when idle, thereby optimizing resource utilization.[1]

A notable advantage of OpenFaaS, distinguishing it from many cloud-specific FaaS offerings like AWS Lambda or Google Cloud Functions, is its inherent portability. Functions developed within OpenFaaS are not confined to a single cloud provider; they can operate seamlessly across any cloud infrastructure or even on-premises deployments without requiring modifications to the underlying code or its configuration.[2] This flexibility empowers organizations to avoid vendor lock-in and maintain consistent deployment strategies across diverse environments.

B. Core Concepts and Architecture

The robust architecture of OpenFaaS is composed of several interconnected components, each playing a vital role in the management and execution of serverless functions.[3] Understanding these core elements is fundamental to effectively utilizing the platform.

1. API Gateway / UI Portal

The API Gateway serves as the primary external entry point for all deployed functions within an OpenFaaS cluster. Beyond routing incoming requests to the appropriate functions, it is instrumental in collecting Cloud Native metrics, which are then integrated with Prometheus for monitoring. A key responsibility of the API Gateway is its dynamic ability to scale functions based on demand, achieved by adjusting the service replica counts within the underlying orchestrator, whether that be Docker Swarm (though now deprecated) or Kubernetes.[3]

Complementing its API functionality, the gateway also features a built-in User Interface (UI). This portal provides a convenient browser-based interface for users to invoke functions directly and to create new ones as needed.[3] As a RESTful microservice, the API Gateway also offers accessible Swagger documentation, detailing its endpoints and operations.[3]

2. Function Watchdog (Classic vs. of-watchdog)

The Function Watchdog is a critical component within OpenFaaS, tasked with the essential role of starting and monitoring functions. It operates as an "init process" for function containers, embedding a Golang-based HTTP server capable of handling concurrent requests, managing timeouts, and performing health checks.[4] This design allows virtually any binary to be transformed into a serverless function through the watchdog's interface.[4]
Historically, the **Classic Watchdog** was the standard for all official OpenFaaS templates. Its operational model involved forking a new process for each incoming request, a design choice that offered a high degree of portability for functions.[4] While robust, this "fork-per-request" model could introduce latency due to process startup overhead.
Introduced in October 2017, the **of-watchdog** project emerged as a complementary, and increasingly popular, alternative. It is particularly well-suited for production environments due to its enhanced capabilities. Unlike the Classic Watchdog, of-watchdog offers alternative communication methods beyond standard input/output (STDIO), and crucially, it can operate in an "HTTP mode".[4] This HTTP mode enables the function process to remain "warm" between invocations, significantly reducing the latency associated with repeatedly forking new processes. This characteristic makes of-watchdog ideal for use cases requiring high-throughput, such as streaming applications, or for functions that need to maintain expensive resources like database connections or machine learning models across multiple requests.[4] The choice between the Classic and of-watchdog is not merely a version preference but a fundamental architectural consideration that directly impacts the performance and resource management of functions. For Python functions, especially those with substantial initialization overhead (e.g., loading large ML models or establishing complex database connections), leveraging the of-watchdog's ability to keep processes warm can lead to a significant reduction in cold start times and overall improved responsiveness.

3. Prometheus/Grafana for Observability

OpenFaaS integrates seamlessly with Prometheus and Grafana to provide robust observability capabilities. Prometheus is utilized by the API Gateway to collect Cloud Native metrics from both the core OpenFaaS components and the deployed functions.[3] These metrics encompass various aspects of system and function performance, including invocation rates, error rates, and resource consumption.
Grafana then serves as the visualization layer, presenting these collected metrics through intuitive dashboards. These dashboards offer real-time insights into auto-scaling actions and overall function performance, allowing operators to monitor the system effectively.[2] The integrated observability stack is a core strength of OpenFaaS, providing a comprehensive view of function behavior. This integration allows developers to quickly identify performance bottlenecks, validate scaling configurations, and ensure the health of their Python functions in a production environment. The ability to observe functions in real-time is crucial for

maintaining optimal performance and reliability.

4. OpenFaaS CLI (faas-cli)

The faas-cli is the cornerstone command-line interface for interacting with OpenFaaS. It functions as a RESTful client for the API Gateway, providing a convenient and powerful toolset for developers. With faas-cli, users can perform a wide array of operations, including deploying, creating, building, pushing, invoking, and generally managing their functions from pre-defined templates.[3] Its comprehensive nature simplifies the entire function lifecycle.

C. OpenFaaS Editions (Community, Standard/Pro, faasd)

OpenFaaS offers distinct editions, each tailored to different user needs and deployment scenarios, from individual exploration to enterprise-grade production environments.[1]

- **Community Edition (CE):** This version is freely available for personal use. For commercial applications, it comes with a 60-day usage limit. OpenFaaS CE is typically deployed to Kubernetes and is well-suited for initial exploration, learning, or Proof-of-Concept (PoC) projects.[1]
- **OpenFaaS Standard/Pro (For Enterprises):** This edition is specifically designed for commercial use and production deployments, leveraging Kubernetes as its orchestration backbone. It includes a suite of advanced features such as the OpenFaaS Pro Autoscaler, the Function Builder API, and enhanced support options, making it suitable for demanding enterprise workloads.[1] The availability of advanced autoscaling and builder APIs in the Pro version enables more sophisticated resource management and CI/CD workflows, which are critical for maintaining performance and reliability in production Python function deployments.
- **faasd:** Representing a lightweight and highly portable iteration of OpenFaaS, faasd is designed to run on a single Virtual Machine (VM) without the overhead or complexity of a full Kubernetes cluster. It is free to use on any cloud or on-premises environment and offers a significantly simpler management experience compared to a Kubernetes-based setup.[1] The explicit design of faasd to bypass Kubernetes complexity offers a distinct advantage for users prioritizing simplicity and lower operational overhead. This approach provides a quick entry point for Python developers who may not require the full feature set or steep learning curve associated with Kubernetes.

The existence of these distinct editions highlights a tiered approach to functionality and support, catering to a wide spectrum of user requirements, from hobbyists to large enterprises. This stratification influences deployment choices, the availability of advanced features (e.g., advanced autoscaling is exclusive to OpenFaaS Pro), and the overall operational complexity. For production-grade Python workloads, OpenFaaS Pro on Kubernetes is generally recommended due to its high availability, performance, and security features. Conversely, faasd offers a compelling alternative for smaller, single-VM deployments where simplicity and reduced resource footprint are paramount.

Table 1: OpenFaaS Core Components and Their Roles

Component Name	**Primary Role**	**Key Features**

API Gateway	External entry point for functions, routing, metrics collection, scaling decisions	Built-in UI, RESTful microservice, integrates with Prometheus, scales functions based on demand
Function Watchdog	Starts and monitors functions, provides HTTP interface to function code	Acts as init process, supports concurrent requests, timeouts, health checks, enables any binary as a function
Prometheus	Collects Cloud Native metrics from OpenFaaS components and functions	Time-series database for monitoring, integrated with API Gateway
Grafana	Visualizes metrics collected by Prometheus	Provides dashboards for real-time auto-scaling and function performance monitoring
OpenFaaS CLI (faas-cli)	Command-line tool for managing OpenFaaS and functions	Deploy, create, build, push, invoke, scale, update, delete functions, manage secrets

II. Setting Up Your OpenFaaS Environment

Establishing an OpenFaaS environment is the foundational step before deploying any functions. This process primarily involves installing the faas-cli and then deploying the OpenFaaS platform itself onto a chosen orchestration environment.

A. Installing the OpenFaaS CLI (faas-cli)

The faas-cli is an indispensable tool for managing OpenFaaS installations and deploying functions. Its installation is a prerequisite for most OpenFaaS workflows.[6]
Several methods are available for installing the faas-cli, with arkade being the recommended approach for its speed and ease of use:

- **arkade (Recommended):** This method offers the quickest and most straightforward installation. First, install arkade itself by executing curl -sSL https://get.arkade.dev | sudo -E sh. Once arkade is installed, the faas-cli can be obtained with arkade get faas-cli.[8] arkade is also capable of installing approximately 120 other DevOps and Kubernetes tools, suggesting a broader strategy by OpenFaaS to simplify the setup of related development and operational tools. This focus on a streamlined installer highlights a commitment to improving the developer experience by reducing the initial complexity of toolchain setup.
- **Bash Script:** For Linux and macOS users, a direct bash script can be used: curl -sSL https://cli.openfaas.com | sudo -E sh. The -E flag ensures that any existing HTTP proxy environment variables are passed through to the installation script.[6] For non-root installations, curl -sSL https://cli.openfaas.com | sh will download the binary to the

current directory and provide further instructions.[21]

- **brew (macOS):** Users on macOS can install faas-cli via Homebrew: brew install faas-cli. However, it is important to note that the version available through brew may sometimes lag slightly behind the latest releases, making arkade a more current option.[6]
- **Windows:** Windows users can find the executable directly on the FaaS releases page or utilize Git Bash with the curl command for installation.[20]

The faas-cli can be updated at any time by simply re-running the chosen installation command (e.g., arkade get faas-cli again).[21]

B. Deploying OpenFaaS

Before proceeding with the deployment of OpenFaaS, it is essential to ensure that the chosen underlying infrastructure is ready. This typically involves having a Kubernetes cluster for OpenFaaS CE/Pro or a single virtual machine for faasd, both with active internet access.[8]

1. On Kubernetes (Recommended for Production)

Deploying OpenFaaS on Kubernetes is the recommended approach for production environments due to its scalability and robustness.

- **Prerequisites:** A functional Kubernetes cluster (which can be a single-node setup for development or a multi-node cluster for production), kubectl configured to interact with your cluster, and helm v3 installed are all necessary before beginning the deployment.[8]
- **Deployment Steps (using arkade - recommended):**
  1. **Install arkade:** If not already installed, run curl -sSLsf https://get.arkade.dev/ | sudo sh.[8]
  2. **Install OpenFaaS:** Execute arkade install openfaas to deploy the OpenFaaS core components onto your Kubernetes cluster.[8]
  3. **Forward Gateway Port:** To access the OpenFaaS gateway from your local machine, set up port forwarding: kubectl port-forward -n openfaas svc/gateway 8080:8080 &.[8]
  4. **Login to Gateway:** Set the OPENFAAS_URL environment variable to the gateway's address (e.g., http://127.0.0.1:8080 or http://127.0.0.1:31112 if using NodePort), then log in using echo -n $FAAS_PASSWD | faas-cli login --password-stdin.[8]
  5. **Verify Pod Readiness:** Wait for the OpenFaaS pods and services to become ready, which might take some time as container images are downloaded: kubectl wait --for=condition=available --timeout=600s deployment/gateway -n openfaas.[24]
- **Deployment Steps (using helm):**
  1. **Add OpenFaaS Helm Repository:** helm repo add openfaas https://openfaas.github.io/faas-netes/.[8]
  2. **Deploy the Chart:** Follow the detailed instructions in the official OpenFaaS documentation for deploying the Helm chart, as specific configurations may vary.[8]

The shift in deployment methods from manual kubectl apply commands to more automated tools like arkade or helm signifies a broader move towards more robust and idempotent infrastructure management practices. This evolution helps ensure consistent deployments and reduces the potential for human error. The explicit instruction to wait for pods to become ready also addresses a common challenge in Kubernetes deployments, where components may not be immediately available after being scheduled.

- **Security Note:** For production deployments, it is crucial to enable basic authentication for the OpenFaaS gateway. This is typically achieved by creating a Kubernetes secret: kubectl -n openfaas create secret generic basic-auth --from-literal=basic-auth-user=$FAAS_USER --from-literal=basic-auth-password="$FAAS_PASSWD".[24]

2. Using faasd (Lightweight, Single VM)

faasd offers a streamlined alternative to a full Kubernetes deployment for OpenFaaS. It is a lightweight and portable version designed to run efficiently on a single host, making it a fast and easy-to-manage solution for scenarios where the full complexity of Kubernetes is not required.[1]

- **Prerequisites:** To run faasd, the host system needs containerd (a core container runtime) and CNI plugins (for container networking). While faas-cli is optional for faasd's operation, it is highly recommended for managing functions.[18]
- **Installation (Developer-focused, from source/binaries):** The installation process for faasd is typically more hands-on, often involving building from source or downloading binaries, and configuring it as a systemd service.
    1. **Install Build Packages and Go:** Ensure necessary build tools (like runc, bridge-utils, make) and the Go programming language are installed on your system.[18]
    2. **Clone faasd Repository:** Obtain the faasd source code by cloning its GitHub repository.[18]
    3. **Build or Download:** You can either build faasd from source using make local within the cloned repository, or download pre-compiled binaries suitable for your architecture.[18]
    4. **Install as Systemd Service:** Install faasd as a systemd unit file using the faasd install command. This configures it to run as a background service.[18]
    5. **Enable IP Forwarding:** Crucially, enable IP forwarding on your host to allow containers to access the internet: sudo /sbin/sysctl -w net.ipv4.conf.all.forwarding=1. This setting should also be made permanent (e.g., by adding it to /etc/sysctl.conf).[18]
- **Running faasd:** After installation, faasd components can be started. This typically involves running sudo -i cd /var/lib/faasd; faasd up for the main faasd service and sudo -i cd /var/lib/faasd-provider; faasd provider for the function provider.[18]
- **Access:** The default basic authentication username for faasd is admin. The password is stored in /var/lib/faasd/secrets/basic-auth-user.[18]

The design philosophy behind faasd explicitly aims to circumvent the complexities and overhead of Kubernetes. This provides a simpler, lower-cost deployment model that is highly beneficial for users who do not require a full-fledged orchestrator. For Python developers, faasd offers a rapid and accessible entry point into serverless computing without the steep learning curve often associated with Kubernetes, allowing them to focus more directly on function development.

3. Note on Docker Swarm (Deprecated Status)

It is important to acknowledge that while OpenFaaS was initially tightly integrated with Docker Swarm, the faas-swarm provider is **no longer supported by the community as of December 2020**.[8] This means that new deployments should strongly favor Kubernetes or faasd as the underlying orchestration platform.

Furthermore, OpenFaaS Community Edition images were migrated from Docker Hub to GitHub Container Registry (ghcr.io) due to changes in Docker Hub's rate limits and policies.[25] This migration underscores the challenges faced by open-source projects in maintaining resilient image distribution channels and highlights the importance of adapting to changes in the broader container ecosystem. For users, this implies that relying on deprecated components or outdated image sources can introduce maintenance burdens and potential security vulnerabilities, reinforcing the need to use currently supported deployment targets.

III. Developing Your First Python Function

Developing serverless functions with OpenFaaS, particularly using Python, is streamlined through the use of language templates and a clear function structure.

A. Understanding OpenFaaS Python Templates

OpenFaaS templates are designed to abstract away the underlying Dockerfile, HTTP server setup, and other boilerplate code. This abstraction allows developers to concentrate primarily on writing the business logic of their functions.[2] These templates are readily available from both the official OpenFaaS store and a broader community marketplace.[24] Users can list all available templates using faas-cli template store list and retrieve specific ones locally with faas-cli template store pull <template_name>.[24]

For Python, OpenFaaS provides two primary official templates, each with distinct characteristics regarding dependencies:

- **python3-http (Alpine-based):** This template is built upon Alpine Linux, which is known for its minimal footprint. Consequently, functions built with this template result in the smallest possible image sizes and offer the fastest startup times. It is ideally suited for Python modules that are "pure Python" and do not rely on native (e.g., C/C++) dependencies.[28]
- **python3-http-debian (Debian-based):** In contrast, this template uses a Debian base image. It is specifically recommended for Python modules that require native dependencies, such such as numpy, pandas, or pillow. While Debian images are larger in size, they provide greater compatibility and efficiency when dealing with modules that

necessitate a native build toolchain. This is because Alpine Linux uses musl libc, which is different from glibc that most native Python modules expect.[28] The choice between these two templates is a crucial decision for Python developers, as it directly impacts the built function's image size, build time, and runtime compatibility with complex libraries. Selecting the appropriate template from the outset can prevent frustrating build failures and optimize function performance.

Beyond these standard HTTP-oriented templates, **Python Flask Templates** (e.g., python*-flask* and their -debian variants) are also available. These templates serve as direct replacements for the classic python3 template, integrating the more efficient of-watchdog and leveraging the Flask framework. This combination provides developers with enhanced control over HTTP requests and responses within their Python functions.[32]

Furthermore, OpenFaaS templates allow for **Python Versioning**. Developers can specify a particular Python version, such as 3.12 or 3.10, by utilizing the PYTHON_VERSION build argument. This argument can be set within the stack.yml file or passed directly via the faas-cli build --build-arg command, offering flexibility in managing Python runtime environments.[32]

B. Function Structure: handler.py and requirements.txt

When a new OpenFaaS function is scaffolded using faas-cli new, it generates a standard directory structure that includes the handler.py file and a requirements.txt file.[2] These two files are central to defining the function's logic and its dependencies.

1. The handle Function Signature

The handler.py file is where the core business logic of a Python function resides. This code executes when an incoming request triggers the function.[2]

The standard entry point for a basic OpenFaaS Python function is a def handle(req): function. In this signature, req typically represents the incoming request body as a string.[3] Any output written to STDOUT by this function is captured by the OpenFaaS watchdog and sent back as the HTTP response to the caller.[3]

For more advanced scenarios, particularly with templates like python3-http or python3-flask, the function signature often expands to def handle(event, context):. Here, the event object provides comprehensive details about the HTTP request, including its body, headers, method, query string parameters, and path. The context object, on the other hand, offers function-specific information.[2] This richer signature allows for more granular control and access to request metadata.

Example handler.py (basic):

Python

```python
import requests

def handle(req):
    """
    Handles a basic HTTP request by making an external GET request.
    The input 'req' is treated as a URL.
    """
    try:
        # Example: Make an HTTP GET request to the input URL
        r = requests.get(req, timeout=1)
        # Print to STDOUT, which becomes the function's response
        return f"{req} => {r.status_code}"
    except requests.exceptions.RequestException as e:
        return f"Error processing {req}: {e}"
```

3

Example handler.py (with event/context):

Python

```
def handle(event, context):
    """
    Handles an HTTP request using the event and context objects.
    """
    # Access request body from event
    input_body = event.body.decode('utf-8')
    # Access headers from event
    content_type = event.headers.get("Content-Type", "text/plain")

    response_body = f"Hello from OpenFaaS! You sent: {input_body}"

    return {
        "statusCode": 200,
        "body": response_body,
        "headers": {"Content-Type": content_type}
    }
```

28

2. Managing Python Dependencies with requirements.txt

The requirements.txt file plays a crucial role in defining the Python packages that your function depends on. During the Docker image build process, these packages are automatically installed using pip.[2]
However, relying solely on requirements.txt presents several limitations that developers should

be aware of:

- **Lack of Dependency Resolution:** When pip install -r requirements.txt is executed, it does not perform a true dependency resolution. Packages are installed sequentially, which can lead to version conflicts or incompatibilities that may only manifest at runtime.[34]
- **No Sub-dependency Management:** The requirements.txt file typically lists only top-level dependencies. It lacks an automated mechanism to manage transitive dependencies (dependencies of your dependencies) and their specific versions, making it challenging to track and address potential security vulnerabilities within the entire dependency tree.[34]
- **Manual Updates:** Maintaining requirements.txt requires manual updates. If new packages or updated versions are introduced into a project, developers must manually remember to update this file along with all its associated dependencies. This manual process is prone to human error and can lead to inconsistencies.[34]

These limitations highlight a significant consideration for any Python project, including serverless functions, regarding dependency management. Moving beyond the basic requirements.txt for production-grade Python functions is a recommended practice to enhance security, reproducibility, and overall maintainability.

Alternatives for Robust Dependency Management:

To address the shortcomings of requirements.txt, the Python community has developed more sophisticated tools:

- **Pipenv:** This tool integrates virtual environments and automatically manages project dependencies. It uses a Pipfile to track direct dependencies and a Pipfile.lock to ensure deterministic builds by locking down exact versions of both direct and sub-dependencies. Pipenv also includes built-in security checks for known vulnerabilities.[34]
- **Poetry:** Another powerful tool, Poetry, focuses on comprehensive dependency management and packaging. It uses the pyproject.toml file for standardized project configuration (adhering to PEP 518) and generates a poetry.lock file for consistent environments. Poetry also simplifies the process of building and publishing packages.[34]

Best Practices for Secure Dependency Management:

Regardless of the chosen tool, adhering to the following best practices is crucial for improving project security and reliability:

- **Regularly Update Dependencies:** Utilize the chosen tool's update mechanism (e.g., pipenv update, poetry update) to keep all dependencies current. This helps mitigate known vulnerabilities and ensures access to the latest bug fixes.[34]
- **Audit Dependencies:** Periodically audit your project's dependencies for known security vulnerabilities. Tools like pipenv check or external services such as Snyk can provide valuable insights.[34]
- **Specify Versions Conservatively:** Define dependency versions using ranges that allow for minor bug fixes and security patches while preventing unintended breaking changes from major version updates.[34]

- **Isolate Environments:** Always use virtual environments to isolate project dependencies. This prevents conflicts between different projects and ensures that each function's dependencies are self-contained.[34]

C. Creating a New Python Function with faas-cli new

The faas-cli new command is the primary method for scaffolding a new OpenFaaS function based on a chosen template. This command simplifies the initial setup by generating the necessary files and directory structure.
To create a new Python function, the following command is used:
faas-cli new --lang <template_name> <function_name>
--prefix="<your_docker_username>".24
The --prefix parameter is crucial as it sets the Docker image prefix, which typically corresponds to your Docker Hub account or another container registry username. For instance, if your Docker Hub username is myuser, the command would be faas-cli new --lang python3-http hello-world --prefix="myuser".[22] If this prefix is not specified during creation, it must be manually edited in the generated stack.yml file afterward.
Upon execution, this command creates a new directory for your function (e.g., hello-world/) containing the handler.py file, the requirements.txt file, and a stack.yml file. The stack.yml file serves as the primary configuration file for the faas-cli when building, pushing, and deploying your function.[24] It is also possible to add multiple functions to the same stack.yml file by using the --append flag with faas-cli new.[28]

D. Local Testing and Development

OpenFaaS provides robust capabilities for local testing and development, allowing developers to iterate quickly on their Python functions without the overhead of deploying to a remote cluster for every change.
Functions can be tested locally using faas-cli local-run --tag=digest --watch. This command runs the function locally, allowing for rapid iteration and testing of the function's logic. The --watch flag enables hot-reloading, automatically rebuilding and restarting the function when code changes are detected.[28] This feature significantly accelerates the development loop by providing immediate feedback on code modifications.
For more granular debugging, a function's container can be run directly with the watchdog. This approach is particularly useful for testing how the function handles specific HTTP methods and headers. For example, a basic Alpine-based function can be run and debugged with: docker run --name debug-alpine -p 8081:8080 -ti functions/alpine:latest sh # fprocess=date fwatchdog &.[35] This allows developers to interact with the function's HTTP endpoint directly and observe its behavior.
Furthermore, during local development, code can be bind-mounted directly into a running container. This allows for live code changes to be reflected without requiring a full rebuild and redeploy cycle. While highly convenient for rapid prototyping, this method should be used sparingly in production-like environments due to potential inconsistencies between development and deployment environments.[35] The availability of faas-cli local-run and direct

Docker debugging capabilities is a significant advantage for developer productivity. These tools reduce the friction of repeated deployments to a cluster, allowing Python developers to quickly test and refine their functions, which ultimately leads to faster development cycles and improved code quality.

IV. Deploying and Managing Python Functions

Once a Python function is developed and tested locally, the next steps involve packaging it into a container image, pushing it to a registry, and then deploying and managing it within the OpenFaaS environment.

A. Building and Pushing Your Function Image (faas-cli build, faas-cli push)

OpenFaaS functions are fundamentally packaged as Open Container Initiative (OCI) images, commonly known as Docker images. This containerization ensures portability and consistency across different deployment environments.[2]

The faas-cli provides dedicated commands for these crucial steps:

- **faas-cli build**: This command is responsible for building the Docker image for your function. It takes your function's code and dependencies (as specified in requirements.txt) and compiles them into a runnable container image, storing it in your local Docker library.[6]
- **faas-cli push**: After building the image locally, faas-cli push is used to upload this Docker image to a remote container registry. This could be a public registry like Docker Hub, GitHub Container Registry (ghcr.io), or a private enterprise registry. Pushing the image to a remote registry is a mandatory step for deploying functions to a Kubernetes cluster or any remote OpenFaaS instance, as the cluster needs to pull the image from a globally accessible location.[6]

Build Options and Arguments:

The faas-cli offers fine-grained control over the image build process through various options and arguments:

- **--build-option**: This flag enables the inclusion of named sets of native modules during the build. For Python templates, this can be used to include development or debug packages. These options can be specified directly in your stack.yml file or passed via the CLI.[31] This capability is particularly useful for Python functions that require complex compiled libraries, allowing developers to customize the base image and installed packages without manually writing or modifying Dockerfiles. This simplifies the build process for diverse Python function requirements.
- **--build-arg**: This flag allows you to pass custom ARG values directly to the Docker build process. This is useful for injecting specific variables like ADDITIONAL_PACKAGE (to install a single native module) or PYTHON_VERSION (to specify a particular Python runtime version).[31] The --build-arg mechanism provides a flexible way to tailor the build environment for specific Python function needs, such as pinning to an older Python

version for compatibility or including a specific system library required by a Python package.

B. Deploying Functions (faas-cli deploy or faas-cli up)

With the function image built and pushed to a registry, the next step is to deploy it to your OpenFaaS environment.

- **faas-cli deploy**: This command deploys a function (from a pre-built and pushed image) to either a local or a remote OpenFaaS gateway.[6]
- **faas-cli up**: For convenience, faas-cli up combines the build, push, and deploy steps into a single command. This streamlines the development and deployment workflow, especially during iterative development cycles.[6]

1. Image Pull Policy and Updates

Understanding how OpenFaaS handles image updates is crucial for reliable deployments. By default, deployed functions utilize an imagePullPolicy of Always. This policy ensures that the Docker image is pulled from the registry every time a deployment changes, which helps refresh functions that use static image tags (e.g., "latest") during an update.[13]
For OpenFaaS Pro customers, this imagePullPolicy can be configured. For instance, IfNotPresent can be set via Helm chart settings or environment variables for faas-netes, which is particularly useful for local development with Minikube to avoid unnecessary pulls.[13]
Updating Functions: To ensure that changes to your function code are reflected in deployed functions, it is critical to either:

- **Change the Docker image tag:** For each new version of your function, use a distinct image tag (e.g., incrementing from 0.1.0 to 0.1.1 or 0.1.0-a). This explicitly signals to Kubernetes that a new image should be pulled and deployed.[13]
- **Add a checksum label:** Alternatively, you can add a checksum of your source files as a label within your stack.yml file. When this label changes, Kubernetes will detect it as a configuration modification and trigger a new deployment.[42]

The behavior of faas-cli deploy concerning image updates and the imagePullPolicy is a common area of confusion. Understanding imagePullPolicy and the necessity of using new image tags or checksums to trigger updates is vital for establishing reliable CI/CD pipelines and preventing stale function deployments. For Python functions, where code changes are frequent, adopting a robust versioning strategy (e.g., semantic versioning for image tags) and integrating checksums into the CI/CD process are essential best practices for ensuring that the latest code is always deployed.

C. Invoking Functions (Synchronous and Asynchronous)

OpenFaaS functions can be invoked in both synchronous and asynchronous modes, offering flexibility depending on the workload's requirements.

1. faas-cli invoke

The faas-cli invoke command provides a straightforward way to trigger a deployed function from the command line. It reads the request body from standard input (STDIN), allowing for simple text or piped data to be sent to the function. For example, echo "input data" | faas-cli invoke my-python-function will send "input data" to the function.[6] The command also supports sending data from files (--data-binary @filename.txt) or directly from STDIN (--data-binary @-) for more complex inputs.[6]

2. Direct HTTP Calls

Functions can also be invoked directly via HTTP requests to the OpenFaaS gateway.

- **Synchronous Invocations:** For synchronous calls, functions are typically invoked via the route http://gateway:8080/function/{function_name}. Both GET and POST HTTP methods are supported for synchronous calls.[2]
- **Asynchronous Invocations:** For asynchronous calls, the route changes to http://gateway:8080/async-function/{function_name}. Asynchronous invocations exclusively support the POST method.[2]

Functions are designed to retrieve various HTTP request details, including headers, query string parameters, and the request body. They can also set the HTTP response status code, headers, and body for the outgoing response.[2] The availability of both faas-cli invoke and direct HTTP calls, combined with synchronous and asynchronous options, highlights the versatility of OpenFaaS. This flexibility in invocation methods allows for seamless integration of OpenFaaS functions into broader systems via standard HTTP protocols. The asynchronous call mechanism, in particular, provides a robust solution for handling long-running, non-blocking operations, which is a common requirement for many Python-based workloads.

D. Scaling Functions

OpenFaaS is designed for high scalability, incorporating robust auto-scaling capabilities to efficiently manage function replicas based on demand.[1]

1. Automatic Scaling (RPS, Capacity, CPU)

The **OpenFaaS Pro Autoscaler** provides sophisticated horizontal scaling for functions, adjusting the number of replicas between a defined minimum and maximum, and even scaling down to zero when idle.[11] Scaling decisions are based on a target load per replica, which is continuously calculated from the current load metrics.[11]

The Pro Autoscaler supports various scaling modes:

- **rps (Requests Per Second):** This mode is ideal for functions that execute quickly and are designed for high throughput.[11]
- **capacity (In-flight requests/connections):** This mode is more suitable for long-running functions or those with a limited number of concurrent requests they can handle efficiently.[11]
- **cpu (CPU usage):** This mode is tailored for CPU-bound workloads, where scaling decisions are made based on the function's CPU consumption.[11]

A significant feature for cost optimization is **Scale to Zero**, an opt-in capability available in OpenFaaS Pro. This feature automatically scales down idle functions to zero replicas after a configurable zero-duration (e.g., 15 minutes by default), thereby conserving cluster resources and reducing infrastructure costs.[11] The detailed autoscaling labels and modes available in OpenFaaS Pro reveal a highly sophisticated resource management system. This capability allows for highly tailored resource allocation, preventing over-provisioning for Python functions with varying workload characteristics (e.g., I/O-bound vs. CPU-bound tasks), ultimately leading to improved operational efficiency and cost optimization.

2. Manual Scaling with Labels

Beyond automatic scaling, developers can also manually influence scaling behavior by setting minimum and maximum replica counts directly via labels in the stack.yml file or through faas-cli commands.

- com.openfaas.scale.min: This label sets the minimum number of function replicas that will always be maintained. The default value is 1, which is also the lowest configurable value.[11]
- com.openfaas.scale.max: This label defines the maximum number of replicas the function can scale to. The default and maximum value for the Community Edition is 5, while OpenFaaS Pro allows for higher limits (e.g., 20).[11]

For OpenFaaS Community Edition (CE), scaling is primarily managed by a single RPS-based rule, where min and max replicas are the main controls for scaling behavior.[11]

Example CLI command with labels for scaling:

Bash

```bash
faas-cli store deploy nodeinfo \
--label com.openfaas.scale.max=10 \
--label com.openfaas.scale.target=50 \
--label com.openfaas.scale.type=rps \
--label com.openfaas.scale.zero=true \
--label com.openfaas.scale.zero-duration=10m
```

11

This command deploys the nodeinfo function with a maximum of 10 replicas, targeting 50 requests per second, using RPS-based scaling, and enabling scale-to-zero after 10 minutes of inactivity.

Table 2: faas-cli Commands for Function Management

Command	**Description**	**Example Usage (for Python functions where applicable)**
faas-cli new	Creates a new function from a template.	faas-cli new --lang python3-http my-function --prefix="myuser"
faas-cli build	Builds a Docker image for the	faas-cli build -f

	function into the local Docker library.	my-function.yml
faas-cli push	Pushes the Docker image to a remote container registry.	faas-cli push -f my-function.yml
faas-cli deploy	Deploys the function to the OpenFaaS gateway.	faas-cli deploy -f my-function.yml
faas-cli up	Combines build, push, and deploy into a single command.	faas-cli up -f my-function.yml
faas-cli invoke	Invokes a deployed function, sending data via STDIN.	`echo "Hello"
faas-cli list	Lists deployed functions, their status, and invocation counts.	faas-cli list
faas-cli remove	Removes a deployed function from the OpenFaaS gateway.	faas-cli remove my-function
faas-cli secret	Manages secrets for functions (create, update, list, delete).	faas-cli secret create my-api-key --from-literal="your_key"
faas-cli template store pull	Downloads official or community templates.	faas-cli template store pull python3-http

Table 3: OpenFaaS Autoscaling Labels (OpenFaaS Pro)

Label Name	**Description**	**Default Value**	**Example Usage (for faas-cli store deploy)**
com.openfaas.scale.max	Maximum number of replicas to scale to.	20	--label com.openfaas.scale.max=10
com.openfaas.scale.min	Minimum number of replicas to maintain.	1	--label com.openfaas.scale.min=2
com.openfaas.scale.zero	Whether to scale to zero replicas when idle.	false	--label com.openfaas.scale.zero=true
com.openfaas.scale.zero-duration	Idle duration before scaling to zero.	15m	--label com.openfaas.scale.zero-duration=5m
com.openfaas.scale.target	Target load per replica for scaling (e.g., RPS, in-flight requests, milli-CPU).	50	--label com.openfaas.scale.target=100
com.openfaas.scale.target-proportion	Proportion of target load to trigger scaling	0.90	--label com.openfaas.scale.tar

	(e.g., 0.9 for 90%).		get-proportion=0.95
com.openfaas.scale.type	Scaling mode: rps, capacity, or cpu.	rps	--label com.openfaas.scale.type=cpu
com.openfaas.scale.down.window	Go duration to smooth out scaling down (e.g., 5m, 300s).	(Off by default)	--label com.openfaas.scale.down.window=3m

E. Updating Functions (Best Practices for Image Tags)

Ensuring that updates to your Python function code are correctly applied to deployed functions is a critical aspect of continuous delivery. A common pitfall occurs when function updates are not reflected in the deployed instances.
To guarantee that function updates are properly applied, it is crucial to change the Docker image tag for each new version of your function.[13] For example, if your current image is tagged my-function:0.1.0, a new version should be pushed as my-function:0.1.1 or my-function:0.1.0-a. This explicit change in the image tag signals to the underlying Kubernetes orchestrator that a new image needs to be pulled and that a redeployment is necessary.
Alternatively, a robust approach is to add a checksum of your source files as a label within your stack.yml configuration. Kubernetes will detect any change in this label as a modification to the function's configuration, which will then trigger a new deployment of the function.[42] This method provides a more automated way to ensure updates are propagated, especially in CI/CD pipelines.
While the default imagePullPolicy of Always (which pulls the Docker image from the registry every time a deployment changes) can help with updates, explicitly changing image tags or using checksums is a more robust and deterministic strategy.[13] The "function not updating" issue is a common challenge in containerized deployments. Understanding the imagePullPolicy and the necessity of using new image tags or checksums to trigger updates is vital for establishing reliable CI/CD pipelines and preventing stale function deployments. For Python functions, where iterative development and frequent code changes are common, adopting a robust versioning strategy (such as semantic versioning for image tags) and integrating checksum generation into the CI/CD process are essential best practices to ensure that the latest code is consistently deployed and executed.

F. Deleting Functions (faas-cli remove)

When a function is no longer needed, it can be easily removed from the OpenFaaS environment using the faas-cli remove command.
To delete a single function, the command is straightforward: faas-cli remove <function_name>.[6]
If multiple functions are defined within a stack.yml file, the entire set of functions can be removed by specifying the YAML file: faas-cli remove -f <stack_file.yml>.[6] This command will

remove all deployed functions that are listed in the specified configuration file.

V. Advanced Python Function Features

OpenFaaS provides several advanced features that enhance the capabilities of Python functions, including asynchronous invocation, secure secrets management, and flexible environment variable handling.

A. Asynchronous Function Invocation

OpenFaaS includes a built-in asynchronous invocation mechanism, supported by a queue-worker, which is particularly well-suited for long-running tasks or operations that do not require an immediate response to the client.[2]

1. Benefits for Long-Running Tasks

The asynchronous model offers significant advantages for Python functions that involve time-consuming operations:

- **Longer Timeouts:** Asynchronous invocation decouples the client's request from the function's execution. This allows tasks to run for extended periods (minutes or even hours) without the client timing out, as the gateway immediately returns a 202 Accepted response.[2]
- **Retries:** Asynchronous functions are designed to be retried automatically until they succeed or reach a predefined maximum retry limit. This significantly improves the reliability of tasks, especially in environments where transient failures might occur.[2]
- **Decoupling and Parallelism:** This mechanism is excellent for "fanning out" requests, allowing a single trigger to initiate many parallel function executions. For example, processing a large CSV file containing numerous URLs can be efficiently handled by fanning out individual URL processing to separate asynchronous function invocations. This also facilitates chaining functions together to build complex data processing pipelines.[2]
- **Resource Availability:** Asynchronous processing allows work to be queued and processed when resources become available, rather than demanding immediate execution. This can lead to more efficient utilization of cluster resources.[17]

The fundamental advantage of asynchronous invocation is its ability to profoundly impact system design for long-running or batch processing tasks. For Python functions, which might often be involved in data processing, machine learning inference, or interactions with external APIs that can introduce latency, asynchronous invocation becomes a critical architectural pattern. It ensures that the overall application remains responsive and resilient, even when individual tasks take considerable time to complete.

2. Triggering and Callback URLs

To invoke a function asynchronously, the request is directed to the gateway route /async-function/{function_name}. It is important to note that only the POST HTTP method is supported for asynchronous invocations.[2] Upon successful submission, the gateway

immediately returns a 202 Accepted response, indicating that the request has been successfully queued for processing.[14]
For scenarios where a function needs to provide a result back to a client or another service after its asynchronous execution completes, **Callback URLs** can be utilized. By including an X-Callback-Url header in the asynchronous invocation request, the function can register a one-time webhook. This webhook will be triggered upon the function's completion, sending the result to the specified URL. The callback URL can point to another OpenFaaS function or any external HTTP endpoint.[2]
Example curl command for asynchronous invocation with a callback:

Bash

```bash
curl http://127.0.0.1:8080/async-function/figlet \
--data "Hi" \
--header "X-Callback-Url: http://<your-ip>:8888"
```

[14]

When making asynchronous calls from within another function, it is crucial to use the fully qualified domain name for the gateway to ensure proper routing and minimize unnecessary DNS lookups. The recommended format is http://gateway.openfaas.svc.cluster.local:8080/async-function/NAME (or /function/NAME for synchronous calls).[14]

Configuration & Limits

Several configuration options and limits apply to asynchronous functions:

- **Timeouts:** The ack_wait timeout for the queue-worker must be configured to be higher than the longest expected execution time of your function. This ensures that the message is not re-queued before the function has a chance to complete.[14]
- **Concurrency / Parallelism:** The max_inflight option for the queue worker (available in OpenFaaS Pro) controls the number of concurrent function invocations. Increasing this value allows the queue worker to process more messages simultaneously. Additional queue-worker replicas can also be added to further increase the total asynchronous concurrency across the cluster.[14]
- **Dedicated Queues:** For OpenFaaS Enterprise, dedicated queues can be utilized for functions with significantly different execution times. This prevents a single slow task from holding up faster-executing functions in a shared queue.[14]

B. Secrets Management

OpenFaaS provides a unified and secure mechanism for managing confidential data, such as API keys, passwords, or sensitive configuration, across both Kubernetes and faasd environments.[44] This abstraction simplifies the handling of sensitive information for developers.

1. Creating, Updating, Listing, Deleting Secrets (faas-cli secret)

The faas-cli secret commands offer a consistent interface for managing secrets:

- **Create:** Secrets can be created from a literal value using --from-literal="<value>" or from a file using --from-file=<path>. It is also possible to pipe values from standard input. Secrets can be targeted to specific namespaces or OpenFaaS gateways.[44]
  - Example: faas-cli secret create my-api-key --from-literal="your_super_secret_key"
- **Update:** Existing secrets can be updated using similar syntax, either from a new literal value or a new file.[44]
  - Example: faas-cli secret update my-api-key --from-file=/path/to/new_key.txt
- **List:** To view existing secrets, use faas-cli secret list or faas-cli secret ls. This command can also list secrets in alternative namespaces or for specific OpenFaaS instances.[44]
- **Delete:** Secrets can be removed using faas-cli secret remove <secret-name>.[36]

The faas-cli secret commands provide a simplified abstraction over the native secret management capabilities of Kubernetes or Docker Swarm. This means that Python developers do not need to learn orchestrator-specific secret commands, streamlining their workflow and promoting consistency across different deployment targets.

2. Accessing Secrets within Functions (/var/openfaas/secrets/SECRET_NAME)

Regardless of the underlying orchestrator, OpenFaaS makes secrets available to functions at a standard, read-only path: /var/openfaas/secrets/SECRET_NAME. The SECRET_NAME corresponds to the name given to the secret when it was created. This path cannot be remapped, so choosing a descriptive and non-conflicting name for the secret is important.[32] Functions should be designed to read their required secret values from these file paths.

Example Python code snippet for accessing a secret:

Python

```python
import os

def handle(req):
    try:
        # Construct the path to the secret file
        api_key_path = "/var/openfaas/secrets/my-api-key"

        # Read the secret value from the file
        with open(api_key_path, "r") as f:
            api_key = f.read().strip() #.strip() to remove potential newlines
        # Use the API key in your function logic
        # For demonstration, returning a partial key
        return {"statusCode": 200, "body": f"Successfully accessed API key (first 4 chars):
{api_key[:4]}..."}
    except FileNotFoundError:
        return {"statusCode": 500, "body": "Error: Secret 'my-api-key' not found in function
environment."}
    except Exception as e:
        return {"statusCode": 500, "body": f"An unexpected error occurred: {str(e)}"}
```

45

3. Why Avoid Environment Variables for Secrets

OpenFaaS strongly recommends against using environment variables for sensitive data, advocating for the dedicated secret management system instead.

- **Security Risk:** Environment variables are generally not considered secure for confidential data. They can inadvertently leak into logs, be exposed through process listings (ps commands), or be captured in other system traces, making them vulnerable to unauthorized access.[15]
- **Immutability:** Storing secrets directly within Docker images (e.g., using COPY or ADD instructions in a Dockerfile) hard-codes the sensitive data into the image. This makes it difficult to change or update secrets without rebuilding and redeploying the entire image, which is inefficient and prone to errors.[45]

The recommended best practice is to leverage the orchestrator's built-in secret store (accessred via faas-cli secret) for all sensitive information. Environment variables, conversely, should be reserved exclusively for non-confidential configuration data.[15] This clear distinction between secrets and environment variables is a foundational security principle. For Python functions, this means utilizing os.getenv() for general, non-sensitive settings, while sensitive credentials must be read from the /var/openfaas/secrets/ path, ensuring a robust and secure configuration strategy for production deployments.

C. Environment Variables for Configuration

While secrets are designated for confidential data, environment variables serve as an effective mechanism for injecting non-confidential configuration data into OpenFaaS functions. This can include settings like logging verbosity, runtime modes, or non-sensitive URLs for external services.[15]

1. Setting via stack.yml and CLI

Environment variables can be defined directly within the environment: section of your function's stack.yml file. This allows for static configuration that is applied when the function is deployed.[22]

Additionally, environment variables can be loaded from external files using the

environment_file: directive in stack.yml. This promotes better organization for larger sets of configuration parameters.[45]
Example stack.yml snippet with environment variables:

YAML

```
functions:
  my-python-function:
    lang: python3-http
    handler:./my-function
    image: your_docker_user/my-python-function:latest
    environment:
      LOG_LEVEL: DEBUG
      API_ENDPOINT: "https://api.example.com/v1"
    environment_file:
      - env.yml # Refers to an external file for additional environment variables
```

22

Example env.yml (for use with environment_file):

YAML

```
environment:
  redis_hostname: "redis-master.redis.svc.cluster.local"
  redis_port: 6379
  s3_bucket: of-demo-inception-data
```

48

The clear distinction between secrets and environment variables for configuration is a fundamental security principle. This separation reinforces why sensitive data should be handled through dedicated secret management, while general configuration can be passed via environment variables. For Python functions, this translates to using os.getenv() for non-sensitive settings and strictly reading credentials from the /var/openfaas/secrets/ path, thereby ensuring a robust and secure configuration strategy.

2. envsubst-style Substitution

The OpenFaaS stack.yml format supports envsubst-style variable substitution. This powerful feature allows for dynamic values to be injected into the stack.yml file from environment variables during the faas-cli build, push, or deploy operations.[28]
This capability is particularly useful in Continuous Integration/Continuous Deployment (CI/CD)

environments, where different configurations (e.g., distinct Docker registries for development vs. production, or specific version tags derived from build pipelines) need to be applied without modifying the stack.yml file itself.[39]

Example stack.yml with envsubst substitution:

YAML

```
functions:
  url-ping:
    lang: python
    handler:./sample/url-ping
    image: ${DOCKER_USER:-exampleco}/faas-url-ping:${VERSION:-latest}
```

39

In this example, ${DOCKER_USER} and ${VERSION} will be replaced by the values of the corresponding environment variables at the time of faas-cli execution. If the environment variables are not set, the default values (exampleco and latest, respectively) will be used.
The integration of envsubst-style variable substitution into stack.yml is a powerful feature for automating deployments and promoting Infrastructure as Code (IaC) principles. This allows Python functions to be built and deployed consistently across various environments (local, staging, production) by simply adjusting environment variables. This approach significantly reduces manual configuration, minimizes the potential for human error, and facilitates a more streamlined CI/CD pipeline.

D. Customizing Python Templates

For scenarios that require a highly specific or unique Python runtime environment, OpenFaaS allows developers to create custom templates. The most straightforward method for creating a custom template is to copy and then modify an existing official template.[29]
A custom template fundamentally requires two main components:

- A template.yml file: This file defines the language name for the template and can include an optional welcome message that is displayed when a new function is scaffolded from it.[29]
- A Dockerfile: This is the core of the custom template, providing full control over the function's base image, the installation of system packages, and the overall runtime environment setup. The Dockerfile is also responsible for configuring and integrating the OpenFaaS watchdog.[29]

For Python functions, modifying the Dockerfile within a custom template offers extensive flexibility beyond what requirements.txt can provide. Developers can:

- **Specify exact Python versions:** Pin to a specific Python version not readily available in standard templates.
- **Install system libraries:** Include necessary system-level dependencies required by

Python packages (e.g., libraries for numpy or pandas that rely on C extensions).

- **Build custom Python environments:** Pre-install complex Python packages or configure specific virtual environments directly into the image.
- **Integrate custom build steps:** Add any custom build logic or pre-compilation steps required for specialized Python dependencies.[32]

The ability to create and customize templates, particularly by modifying their Dockerfiles, offers ultimate flexibility. This deep customization is especially valuable for Python functions with highly specific or complex dependency requirements, such as those involving pre-compiled scientific libraries or custom C extensions. Custom templates empower developers to precisely control the build environment, overcoming the limitations of standard requirements.txt and ensuring complete reproducibility and compatibility for their functions.

Table 4: OpenFaaS Secret Management Commands

Command	Purpose	Example Syntax
faas-cli secret create	Creates a new secret.	faas-cli secret create my-db-password --from-literal="supersecurepassword"
faas-cli secret update	Updates an existing secret.	faas-cli secret update my-db-password --from-file=/path/to/new_password.txt
faas-cli secret list	Lists all secrets in the current namespace.	faas-cli secret list
faas-cli secret remove	Deletes a specified secret.	faas-cli secret remove my-db-password

VI. Best Practices and Troubleshooting

Deploying and operating OpenFaaS functions, particularly Python functions, in a production environment necessitates adherence to specific best practices across security, performance, and high availability. Understanding common pitfalls and effective troubleshooting techniques is equally vital for maintaining a robust and reliable serverless platform.

A. Production Best Practices

Effective production deployments of OpenFaaS require careful configuration and operational strategies.[15]

1. Security

- **Non-root user for functions:** It is strongly recommended to configure functions to run under a non-root user. This principle of least privilege significantly mitigates the impact of potential vulnerabilities within the function code or its dependencies, as it restricts the actions a compromised function can perform within the container and on the host system.[15]

- **Read-only filesystem:** Enabling a read-only filesystem for functions enhances security by preventing unauthorized writes to the container's filesystem. While the /tmp/ directory remains writable for temporary file storage, the core function environment is protected.[15]
- **Appropriate use of secrets and environment variables:** A clear distinction must be made between confidential data and non-confidential configuration. Secrets (managed via faas-cli secret) should be exclusively used for sensitive information such as API keys, database credentials, or private IP addresses. Environment variables, conversely, are suitable for non-confidential settings like logging levels, runtime modes, or non-sensitive URLs.[15] This separation is a critical security principle, as environment variables can easily leak into logs or process lists.
- **Encrypting Secret Data at Rest:** For enhanced security, Kubernetes supports encrypting secrets at rest. While optional, this configuration is highly recommended to ensure that sensitive data is encrypted when stored on disk.[15]
- **NetworkPolicy:** Implementing Kubernetes NetworkPolicies is crucial to restrict communication between function namespaces and the core OpenFaaS components (the control plane). This prevents unauthorized or unintended access by user-deployed functions to sensitive platform services, requiring a network driver that supports NetworkPolicy (e.g., Weave Net, Calico).[15] The combination of these security practices forms a multi-layered defense-in-depth strategy, minimizing the attack surface and mitigating the impact of potential vulnerabilities within Python functions or their dependencies.

2. Performance

Optimizing the performance of OpenFaaS and its functions involves tuning various parameters across the system.[15]

- **Configure timeouts:** Timeouts for the gateway, individual functions, and any intermediate components like Ingress controllers or cloud Load Balancers must be carefully configured to match the expected execution times of your workloads. Mismatched timeouts can lead to premature request termination or unnecessary retries.[15]
- **Watchdog choice:** For high-throughput and low-latency operations, especially with Python functions, the of-watchdog is generally preferred over the classic watchdog. Its ability to keep function processes warm significantly reduces cold start latency, which is critical for responsive applications.[15]
- **Reduce DNS lookups:** When functions need to call the OpenFaaS gateway or other functions, using fully qualified domain names (FQDNs) like http://gateway.openfaas.svc.cluster.local:8080/function/function-name can significantly reduce DNS resolution overhead. This optimization minimizes dozens of DNS lookups per outgoing HTTP call, improving overall request latency.[15]
- **Health-check probes:** Utilize HTTP probes for function health checks instead of exec

probes. HTTP probes are generally more CPU-efficient, contributing to better overall cluster performance.[15] These performance tuning recommendations highlight the importance of understanding the entire request path in a serverless environment. For Python functions, which can be sensitive to startup times or external network calls, these optimizations directly translate to lower latency and higher throughput, ensuring efficient operation.

3. High Availability (HA)

Ensuring high availability for OpenFaaS components is paramount for continuous operation of Python functions in a production environment, preventing single points of failure.[15]

- **OpenFaaS gateway:** Deploy a minimum of three replicas for the OpenFaaS gateway. Implement Topology Spread Constraints to ensure these replicas are distributed across different nodes, maintaining availability even if a node fails.[15]
- **OpenFaaS queue-worker:** Run at least two replicas of the queue worker, which is responsible for processing asynchronous invocations. Each worker can handle hundreds of concurrent invocations, and multiple replicas provide redundancy and increased processing capacity.[15]
- **NATS JetStream:** The default in-memory storage for NATS JetStream means messages can be lost if the Pod crashes. For production, it is recommended to install NATS separately with a Persistent Volume and high-availability configuration, rather than relying on the default ephemeral setup.[15]
- **Prometheus:** While the default Prometheus instance is ephemeral, if long-term retention of metrics is required, a second Prometheus instance should be run to "federate" or scrape metrics from the internal one. This ensures historical data is preserved.[15]
- **Ingress & TLS:** For public-facing installations, configure an IngressController and TLS (Transport Layer Security) to encrypt traffic between clients and the OpenFaaS Gateway. This protects data in transit.[15]
- **Helm Charts & GitOps:** Utilize Helm charts for flexible and customizable deployments of OpenFaaS. Integrate with GitOps tools like Flux and ArgoCD for declarative management of your deployments, ensuring that the desired state of your infrastructure and functions is consistently maintained from a version-controlled repository.[8]

B. Common Pitfalls and Troubleshooting

Even with best practices in place, issues can arise. Understanding common pitfalls and effective troubleshooting techniques is crucial for maintaining OpenFaaS functions.

1. OpenFaaS Not Starting

- **Symptoms:** Core OpenFaaS deployments (e.g., gateway) show 0/1 replicas, restarts, or errors when checking kubectl get deploy -n openfaas.[43]
- **Common Causes:**
    - Missing gateway password: The basic authentication secret for the gateway might

    not have been created.[43]
  - Networking issues: Core components like NATS and Prometheus might be crashing due to network problems preventing inter-container communication or DNS resolution.[43]
  - Insufficient cluster resources: The cluster may lack sufficient CPU or memory for all OpenFaaS services to start.[43]
- **Troubleshooting Steps:**
  - Inspect deployment status: kubectl get deploy -n openfaas.
  - Describe failing deployments: kubectl describe deploy/gateway -n openfaas.
  - Check logs of failing pods: kubectl logs -n openfaas deploy/gateway.

2. Function Not Starting / Updating

- **Symptoms:** Function replicas show 0/1 or errors, or changes to function code are not reflected after deployment.[42]
- **Common Causes:**
  - Private registry issues: If using a private container registry, image pull secrets might be missing or misconfigured.[43]
  - Missing secrets: The function might require a secret that has not been created in OpenFaaS.[43]
  - Code errors: Errors in the Python function code itself can cause the container to crash on startup.
  - Image caching: Kubernetes might be using a cached image if the image tag has not changed, preventing the new code from being pulled.[13]
- **Troubleshooting Steps:**
  - Check function replica status: kubectl get deploy -n openfaas-fn.
  - Inspect function logs: faas-cli logs <function_name>.
  - Verify image pull secrets configuration.
  - Ensure a new image tag is used for each update, or add a checksum label to your stack.yml to force redeployment.[42] The recurring issues with function startup and updates highlight common challenges in container lifecycle management within Kubernetes. For Python functions, ensuring all requirements.txt dependencies are correctly installed and that the image is pulled correctly are frequent culprits.

3. Timeouts and Empty Bodies

- **Symptoms:** Function invocations time out, or the function receives an empty HTTP request body.[35]
- **Common Causes:**
  - Insufficient timeout configuration: Timeouts for the gateway, function, or queue-worker (for async calls) are too low for the function's execution time.[35]
  - Missing Content-type header: The client invoking the function might not be sending a Content-type header, leading to an empty body being passed to the function.[43]

    - Legacy HTTP servers: If using a legacy HTTP server (e.g., WSGI) with of-watchdog, the http_buffer_req_body: true environment variable might be needed to properly buffer the request body.[43]
- **Troubleshooting Steps:**
    - Review and increase timeouts across the entire invocation path (client, gateway, function).
    - Ensure clients send appropriate Content-type headers (e.g., application/json, text/plain).
    - Check gateway, provider, and queue-worker logs for timeout messages. Timeouts and empty bodies are classic distributed system problems. Understanding and configuring timeouts across the entire stack (client, gateway, function, external services) and ensuring correct HTTP headers are sent is crucial for reliable operation of Python functions, especially those interacting with other services.

4. Incorrect Passwords / Unauthorized Access

- **Symptoms:** Users are unable to log in to the OpenFaaS UI or faas-cli reports unauthorized access errors.[16]
- **Common Causes:**
    - Missing or incorrect basic authentication secret for the gateway.
    - SSO configuration issues (for OpenFaaS Pro): Identity provider not registered, no matching IAM roles/policies, or invalid audience in JWT tokens.[16]
- **Troubleshooting Steps:**
    - Verify the basic authentication secret is correctly created and mounted.
    - Check dashboard logs for SSO-related errors (kubectl logs -n openfaas deploy/dashboard).
    - Ensure IAM roles and policies are correctly applied and match the authenticated user.[16]

5. Function Takes Too Long to Start Up

- **Symptoms:** Functions fail to start due to exceeding startup time limits, often seen with Python functions that have large dependency trees or heavy initialization logic.[43]
- **Common Causes:**
    - Large image size or complex dependencies requiring extensive installation during startup.
    - Heavy initialization logic within the handler.py that runs before the function is ready to serve requests.
- **Troubleshooting Steps:**
    - Optimize function startup: Reduce image size, lazy load dependencies, or pre-warm resources using the of-watchdog.[4]
    - Configure custom HTTP health-checks that accurately reflect when the function is ready to serve traffic.[43]
    - Extend the health-check window to allow for longer startup times if necessary.[43]

Long startup times can negate the benefits of serverless. For Python functions, this often relates to large dependency trees or heavy initialization logic. Optimizing the function's startup and correctly configuring health checks are key to ensuring responsiveness.

Table 5: Common Troubleshooting Scenarios and Solutions

Problem Description	Symptoms	Common Causes	Troubleshooting Steps/Commands
OpenFaaS Not Starting	0/1 replicas, restarts, or errors in kubectl get deploy -n openfaas	Missing gateway password, networking issues (NATS/Prometheus), insufficient cluster resources	kubectl describe deploy/gateway -n openfaas, kubectl logs -n openfaas deploy/gateway
Function Not Starting / Updating	0/1 function replicas, changes not reflected, errors in kubectl get deploy -n openfaas-fn	Private registry issues, missing secrets, code errors, image caching	faas-cli logs <function_name>, ensure new image tag or checksum on update
Function Timing Out	Function invocations fail with timeout errors	Insufficient timeouts (gateway, function, queue-worker), missing Content-type header, legacy HTTP server issues	Increase timeouts, ensure Content-type header, check http_buffer_req_body for of-watchdog
Incorrect Passwords / Unauthorized Access	Unable to login to UI/CLI, "Not Authorized" messages	Incorrect basic auth secret, SSO misconfiguration (IAM roles, policies, issuer, audience)	Verify secrets, check dashboard logs, review IAM configurations
Function Takes Too Long to Start Up	Function fails to start, health checks fail	Large image size, heavy initialization logic, slow dependency loading	Optimize function code for faster startup, configure custom HTTP health checks, extend health-check window

VII. Conclusions

This comprehensive documentation has explored the OpenFaaS platform, with a particular emphasis on its utility and best practices for Python function development and deployment. OpenFaaS offers a robust and portable serverless solution, distinguishing itself from cloud-specific alternatives by enabling consistent function execution across diverse

environments, from Kubernetes clusters to single-VM faasd instances.
The architectural components, including the API Gateway, Function Watchdog (with its critical of-watchdog variant for performance optimization), and integrated observability with Prometheus/Grafana, collectively provide a powerful framework for managing event-driven workloads. The faas-cli acts as the central command-line interface, simplifying the entire function lifecycle from creation to deletion.
For Python developers, OpenFaaS provides tailored templates (python3-http for pure Python, python3-http-debian for native dependencies) that abstract away container complexities, allowing focus on business logic. The handle function signature and requirements.txt form the core of Python function structure, though advanced dependency management tools like Pipenv and Poetry are recommended for production-grade security and reproducibility. The platform's support for envsubst-style variable substitution in stack.yml further enhances CI/CD automation and environment-specific configuration.
Advanced features such as asynchronous invocation are crucial for handling long-running or batch processing Python tasks, improving system responsiveness and reliability through queuing and retries. OpenFaaS's unified secrets management system, which strongly advocates against using environment variables for sensitive data, promotes secure handling of confidential information by making secrets available through a standardized filesystem path.
Finally, successful production deployment of OpenFaaS functions hinges on adhering to best practices in security (non-root users, read-only filesystems, proper secret usage), performance (timeout tuning, of-watchdog utilization, DNS optimization), and high availability (multiple replicas, persistent storage for critical components). Understanding common troubleshooting scenarios, from function startup failures to timeout issues, is essential for maintaining operational stability.
In essence, OpenFaaS provides a flexible, powerful, and developer-friendly platform for serverless Python functions. By leveraging its architectural strengths, adhering to recommended practices, and understanding its operational nuances, organizations can build and deploy highly scalable, secure, and resilient event-driven applications.

Works cited

1. OpenFaaS: Introduction, accessed on May 26, 2025, https://docs.openfaas.com/
2. How to Build & Integrate with Functions using OpenFaaS ..., accessed on May 26, 2025, https://www.openfaas.com/blog/integrate-with-openfaas/
3. Architecture - OpenFaaS, accessed on May 26, 2025, https://ericstoekl.github.io/faas/architecture/
4. Watchdog - OpenFaaS, accessed on May 26, 2025, https://docs.openfaas.com/architecture/watchdog/
5. Watchdog - OpenFaaS, accessed on May 26, 2025, https://ericstoekl.github.io/faas/developer/watchdog/
6. openfaas/faas-cli: Official CLI for OpenFaaS - GitHub, accessed on May 26, 2025, https://github.com/openfaas/faas-cli
7. openfaas/faasd: Lightweight and portable version of ... - GitHub, accessed on

May 26, 2025, https://github.com/openfaas/faasd
8. OpenFaaS CE - OpenFaaS, accessed on May 26, 2025, https://docs.openfaas.com/deployment/kubernetes/
9. OpenFaaS Pro function builder API examples - GitHub, accessed on May 26, 2025, https://github.com/openfaas/function-builder-examples
10. Building Blocks for Source to URL with OpenFaaS, accessed on May 26, 2025, https://www.openfaas.com/blog/source-to-url/
11. Autoscaling - OpenFaaS, accessed on May 26, 2025, https://docs.openfaas.com/architecture/autoscaling/
12. Overview - OpenFaaS, accessed on May 26, 2025, https://docs.openfaas.com/openfaas-pro/iam/overview/
13. openfaas/faas-netes: Serverless Functions For Kubernetes - GitHub, accessed on May 26, 2025, https://github.com/openfaas/faas-netes
14. Async - OpenFaaS, accessed on May 26, 2025, https://docs.openfaas.com/reference/async/
15. Production - OpenFaaS, accessed on May 26, 2025, https://docs.openfaas.com/architecture/production/
16. Troubleshooting Identity and Access Management (IAM) - OpenFaaS, accessed on May 26, 2025, https://docs.openfaas.com/openfaas-pro/iam/troubleshooting/
17. Asynchronous Functions - OpenFaaS, accessed on May 26, 2025, https://ericstoekl.github.io/faas/developer/asynchronous/
18. faasd/docs/DEV.md at master · openfaas/faasd - GitHub, accessed on May 26, 2025, https://github.com/openfaas/faasd/blob/master/docs/DEV.md
19. faasd walk-through with cloud-init and Multipass - YouTube, accessed on May 26, 2025, https://www.youtube.com/watch?v=WX1tZoSXy8E
20. First Python Function - OpenFaaS, accessed on May 26, 2025, https://docs.openfaas.com/tutorials/first-python-function/
21. Installation - OpenFaaS, accessed on May 26, 2025, https://docs.openfaas.com/cli/install/
22. Get storage for your functions with Python and MongoDB - OpenFaaS, accessed on May 26, 2025, https://www.openfaas.com/blog/get-started-with-python-mongo/
23. Use OpenFaaS with AKS on Windows Server - AKS enabled by Azure Arc | Microsoft Learn, accessed on May 26, 2025, https://learn.microsoft.com/en-us/azure/aks/aksarc/openfaas
24. OpenFaaS* — Documentation for Clear Linux* project, accessed on May 26, 2025, https://www.clearlinux.org/clear-linux-documentation/tutorials/openfaas.html
25. How do changes to the Docker Hub affect OpenFaaS?, accessed on May 26, 2025, https://www.openfaas.com/blog/how-does-docker-hub-affect-openfaas/
26. openfaas/faas-swarm: OpenFaaS provider for Docker Swarm - GitHub, accessed on May 26, 2025, https://github.com/openfaas/faas-swarm
27. Docker Swarm - OpenFaaS, accessed on May 26, 2025, https://ericstoekl.github.io/faas/deployment/swarm/
28. How to Build and Scale Python Functions with OpenFaaS, accessed on May 26, 2025, https://www.openfaas.com/blog/build-and-scale-python-function/

29. Custom - OpenFaaS, accessed on May 26, 2025, https://docs.openfaas.com/languages/custom/
30. Create functions - OpenFaaS, accessed on May 26, 2025, https://docs.openfaas.com/cli/templates/
31. OpenFaaS YAML - OpenFaaS, accessed on May 26, 2025, https://docs.openfaas.com/reference/yaml/
32. openfaas/python-flask-template: HTTP and Flask-based ... - GitHub, accessed on May 26, 2025, https://github.com/openfaas/python-flask-template
33. workshop/lab3.md at master · openfaas/workshop · GitHub, accessed on May 26, 2025, https://github.com/openfaas/workshop/blob/master/lab3.md
34. Moving Away from requirements.txt for More Secure Python ..., accessed on May 26, 2025, https://www.pullrequest.com/blog/moving-away-from-requirements-txt-for-more-secure-python-dependencies/
35. Troubleshooting - OpenFaaS, accessed on May 26, 2025, https://ericstoekl.github.io/faas/troubleshooting/
36. Morning coffee with the OpenFaaS CLI - Alex Ellis' Blog, accessed on May 26, 2025, https://blog.alexellis.io/quickstart-openfaas-cli/
37. Building OpenFaaS Serverless function to detect weather using OpenWeatherMap and Python - Faizan Bashir, accessed on May 26, 2025, https://www.faizanbashir.me/building-openfaas-serverless-function-to-detect-weather-using-openweathermap-and-python
38. Build functions - OpenFaaS, accessed on May 26, 2025, https://docs.openfaas.com/cli/build/
39. faas-cli/README.md at master · openfaas/faas-cli · GitHub, accessed on May 26, 2025, https://github.com/openfaas/faas-cli/blob/master/README.md
40. How to Convert Scripts & HTTP Servers to Serverless Functions | OpenFaaS, accessed on May 26, 2025, https://www.openfaas.com/blog/program-to-function/
41. Private Registries - OpenFaaS, accessed on May 26, 2025, https://docs.openfaas.com/reference/private-registries/
42. Question about deploying updated functions · Issue #1701 · openfaas/faas - GitHub, accessed on May 26, 2025, https://github.com/openfaas/faas/issues/1701
43. Troubleshooting - OpenFaaS, accessed on May 26, 2025, https://docs.openfaas.com/deployment/troubleshooting/
44. Manage secrets - OpenFaaS, accessed on May 26, 2025, https://docs.openfaas.com/cli/secrets/
45. Unifying Secrets for OpenFaaS | OpenFaaS - Serverless Functions ..., accessed on May 26, 2025, https://www.openfaas.com/blog/unified-secrets/
46. Question: Secrets as environment variables? · Issue #153 · openfaas/faas-netes - GitHub, accessed on May 26, 2025, https://github.com/openfaas/faas-netes/issues/153
47. stack package - github.com/openfaas/faas-cli/stack - Go Packages, accessed on May 26, 2025, https://pkg.go.dev/github.com/openfaas/faas-cli/stack
48. openfaas/python-fan-in-example: Python code example for ... - GitHub, accessed

on May 26, 2025, https://github.com/openfaas/python-fan-in-example
49. How to package OpenFaaS functions with Helm, accessed on May 26, 2025, https://www.openfaas.com/blog/howto-package-functions-with-helm/
50. Performance - OpenFaaS, accessed on May 26, 2025, https://docs.openfaas.com/architecture/performance/