"ALEXANDRU-IOAN CUZA" UNIVERSITY OF IAȘI

FACULTY OF COMPUTER SCIENCE

BACHELOR'S THESIS

AutoDeployr: Deployment automation system

proposed by

Mihnea-Tudor Zară

Session: july, 2025

Scientific Coordinator

Assoc. Prof. Andrei Panu

”ALEXANDRU-IOAN CUZA” UNIVERSITY OF IAȘI

FACULTY OF COMPUTER SCIENCE

AutoDeployr: Deployment automation system

Mihnea-Tudor Zară

Session: july, 2025

Scientific Coordinator

Assoc. Prof. Andrei Panu

Avizat,
Îndrumător lucrare de licență,
Asoc. Prof. Andrei Panu.
Data: ..... Semnătura: .....

DECLARAȚIE privind autenticitatea conţinutului lucrării de licenţă

Subsemnatul **Zară Mihnea-Tudor** domiciliat în **România, jud. Iași, loc. Valea Lupului, str. Zaharia Stancu nr. 9**, născut la data de **8 aprilie 2003**, identificat prin CNP **5030408226726**, absolvent al Universităţii „Alexandru Ioan Cuza" din Iaşi, Facultatea de Informatică specializarea informatică în limba engleză, promoția 2025, declar pe propria răspundere, cunoscând consecinţele falsului în declaraţii în sensul art. 326 din Noul Cod Penal şi dispoziţiile Legii Educaţiei Naţionale nr. 1/2011 art.143 al. 4 si 5 referitoare la plagiat, că lucrarea de licență cu titlul **AutoDeployr: Deployment automation system** elaborată sub îndrumarea domnului Asoc. Prof. Andrei Panu, este autentică, îmi aparține şi îmi asum conţinutul său în întregime.

De asemenea, declar că sunt de acord ca lucrarea mea de licenţă să fie verificată prin orice modalitate legală pentru confirmarea autenticității, consimțind inclusiv la introducerea conţinutului său într-o bază de date în acest scop. Declar că lucrarea de faţă are exact acelaşi conţinut cu lucrarea în format electronic pe care profesorul îndrumător a verificat-o prin intermediul software-ului de detectare a plagiatului.

Am luat la cunoştinţă despre faptul că este interzisă comercializarea de lucrări ştiinţifice în vederea facilitării falsificării de către cumpărător a calităţii de autor al unei lucrări de licenţă, de diplomă sau de disertaţie şi în acest sens declar pe proprie răspundere că lucrarea de faţă nu a fost copiată, ci reprezintă rodul cercetării pe care am întreprins-o.

Data: ..... Semnătura student: .....

Declarație de consimțământ

Prin prezenta declar că sunt de acord ca Lucrarea de licență cu titlul **AutoDeployr: Deployment automation system**, codul sursă al programelor și celelalte conținuturi (grafice, multimedia, date de test etc.) care însoțesc această lucrare să fie utilizate în cadrul Facultății de Informatică.

De asemenea, sunt de acord ca Facultatea de Informatică de la Universitatea „Alexandru Ioan Cuza" din Iași, să utilizeze, modifice, reproducă și să distribuie în scopuri necomerciale programele-calculator, format executabil și sursă, realizate de mine în cadrul prezentei lucrări de licență.

Absolvent **Mihnea-Tudor Zară**

Iași, data: ..... Semnătura:

.....

Contents

Introduction

Serverless computing promised the simplification of deployment by eliminating server management [1], but instead, it created new complexity with configuration and platform specific knowledge. Getting applications to run on AWS Lambda needs configuring API Gateway, IAM roles and writing deployment scripts, tasks that don't have anything to do with coding application logic. Google Cloud Functions fails builds when dependency resolution does not work and Azure Functions forces developers to choose a hosting model before they can deploy.

Serverless was supposed to let developers focus on business logic instead of infrastructure, yet the deployment process itself has become a specialized skill requiring deep platform knowledge. Many developers spend more time fighting deployment configurations than writing the actual functions they want to deploy. This creates a barrier that contradicts the fundamental promise of serverless computing.

The problem extends beyond individual frustration. Companies adopting serverless often need dedicated DevOps teams just to handle deployment pipelines, negating much of the operational simplicity that made serverless attractive in the first place. Teams that could benefit from serverless's scalability and cost model often stick with traditional deployment methods simply because they're more predictable and require less specialized knowledge.

This thesis presents AutoDeployr, an automatic serverless deployment platform that converts backend applications into serverless functions. Instead of making developers decompose applications and learn specific platform configurations, AutoDeployr analyzes applications using Abstract Syntax Trees [2], extracts individual functions and handles the entire deployment.

The platform addresses what should be a straightforward process but has become surprisingly complex. Most existing applications aren't written as collections of independent functions—they're monolithic web applications with routes, controllers,

and shared dependencies. Breaking these apart manually requires understanding both the application's internal structure and the target platform's deployment requirements. AutoDeployr eliminates this manual process entirely.

The platform solves a big technical challenge, that is automatic dependency resolution when breaking applications into individual endpoints. When AutoDeployr extracts a single route from a Flask app with 10 endpoints, it needs to find which libraries that specific route uses. If only one route needs NumPy for data processing, only that function's container will include NumPy, as the other ones should not have unnecessary dependencies that might waste resources.

This dependency challenge shows a broader problem with current serverless platforms. They assume developers will structure their code specifically for serverless deployment, but most real applications weren't written that way. Existing codebases have complex dependencies, shared utilities, and framework-specific patterns that don't map cleanly to individual functions. AutoDeployr bridges this gap by understanding these patterns and handling the transformation automatically. AutoDeployr supports multiple deployment approaches to help with any workflows. Developers can upload existing applications as ZIP files, write functions directly in the web interface, deploy from GitHub repositories or describe functionalities in natural language and get working code with the AI generation service. These multiple approaches reflect how teams actually work. Some developers have existing applications they want to modernize. Others prefer to prototype quickly with minimal setup. Some teams use version control workflows that should integrate seamlessly with deployment. The platform accommodates these different preferences without forcing teams to change their development processes.

This work eliminates deployment complexity that can be a barrier to serverless adoption. The research demonstrates that code analysis can automatically decompose applications while maintaining performance. If serverless computing offers advantages for modern development, developers should have those advantages without first becoming experts in platform-specific deployment procedures.

The thesis is structured in main 3 chapters that describe the AutoDeployr system. Chapter 1 presents the problem of serverless deployment complexity across cloud providers and introduces the platform. It details the 4 deployment methods and function management capabilities and analyzes existing serverless platforms and their limitations. Chapter 2 covers the technical implementations of the backend and the fron-

tend, focuses on the code analysis system for function extraction, the deployment pipeline with Docker containerization, OLLAMA integration for generating code, database design and security mechanisms that include JWT authentication and function API keys. Chapter 3 shows the platform's practical value through 4 real world scenarios: mobile app backend development, rapid prototyping with AI generation, multi framework team collaboration, and healthcare system integration, showing how AutoDeployr addresses different workflows and organizational needs. The conclusion summarizes the research findings and limitations that include scalability and security, and presents future improvements that are needed for production deployment.

Contributions

The work for AutoDeployr makes several contributions to the field of serverless automatic deployment. The primary contribution is the zero-configuration automatic deployment. Unlike existing platforms where developers need to write deployment scripts, configuration files, and understand concepts that are platform specific, this system automatically handles everything from source code to HTTP endpoints without requiring any other steps. Second contribution is the code analysis system that uses Abstract Syntax Trees to extract HTTP endpoints from web applications. Unlike existing platforms that make developers manually write serverless functions for each endpoint, this system uses AST to understand the app structure for multiple programming languages and frameworks. The Python Flask [3] analyzer uses python's AST [4] module to parse route decorators, extract function dependencies, and handle database integrations. The Java Spring Boot [5] analyzer uses JavaParser [6] to process spring annotations and check Maven dependencies, while the PHP Laravel [7] analyzer nikic/php-parser [8] to check the router definitions and controller methods. The analyzers also extract each endpoint's specific dependencies, so there are no unnecessary dependencies in the Docker images. This helps with startup time and resource usage. Another contribution is the integration with OLLAMA [9] code generation, which converts natural language description of features a user wants to implement into working, deployable functions. The system is framework aware and generates code following specific rules.

Chapter 1

Application Description

1.1 Addressed problem

Serverless computing promised a way to make application development simpler by eliminating server management, but deployment became a productivity problem. Platforms like AWS Lambda, Google Cloud Functions and Azure Functions have powerful execution, but getting applications deployed is still a hard task. The main issues come from platform-specific configuration requirements that create unnecessary problems. Each major cloud provider has different deployment approaches. For example, while AWS Lambda offers simple function URL for public endpoints since 2022, private functions still force developers to set up IAM roles and API Gateway with their own billing [10, 11].

The core problem is not just platform complexity, but the lack of intelligent deployment automation. Current serverless platforms require developers to manually decompose applications, configure dependencies, and write deployment scripts. What's needed is a system that can automatically analyze application code, extract deployable functions, and handle the entire deployment pipeline without manual configuration.

The motivation for AutoDeployr comes from a simple observation: if serverless computing is supposed to reduce operational complexity, then deployment itself should not be the source of that complexity. By providing a unified platform that handles deployment, containerization and dependency resolution and ensuring simplicity for both private and public functions, developers should be able to focus on writing application logic rather than fighting with deployment configurations.

1.2 Proposed solution

To address these deployment problems, we built AutoDeployr. The idea came from a simple observation: if serverless computing is supposed to reduce operational complexity, then deployment itself shouldn't be the source of that complexity.

AutoDeployr automatically converts backend applications into containerized serverless functions. Instead of requiring developers to manually break apart their applications and learn platform-specific configurations, the system analyzes the application structure, extracts individual functions, and handles the entire deployment pipeline automatically.

The platform supports four different ways to deploy code, depending on how developers prefer to work. You can upload a ZIP file of existing applications, write functions directly in the web interface for quick prototyping, connect GitHub repositories for version-controlled projects, or describe what you want in natural language and let AI generate the code.

What makes this different from existing deployment tools is that it actually understands your code structure. Most platforms treat your application as a black box meaning they package everything up and hope for the best. AutoDeployr uses Abstract Syntax Tree parsing to analyze Flask routes, Spring Boot controllers, Laravel endpoints, and other framework patterns. This means it can extract exactly which dependencies each function needs, creating lightweight containers instead of bloated ones that include everything.

The deployment process is simple whichever method you chose. You can upload code and the system automatically identifies the framework, extracts the endpoints and builds Docker [12] containers with the right dependencies. Each function gets its own container so dependency conflicts that cause problems on other platforms are eliminated. Current automated deployment tools still require manual configuration and platform-specific knowledge.

1.3 Application functionalities

1.3.1 Registration and login

New users begin by creating an account through the registration interface where they provide basic information like username, email, password [Figure 1.1]. Once this step is completed, they can access the platform by logging in with their credentials, which generates a JWT token for secure session management. The platform also includes password reset functionality.

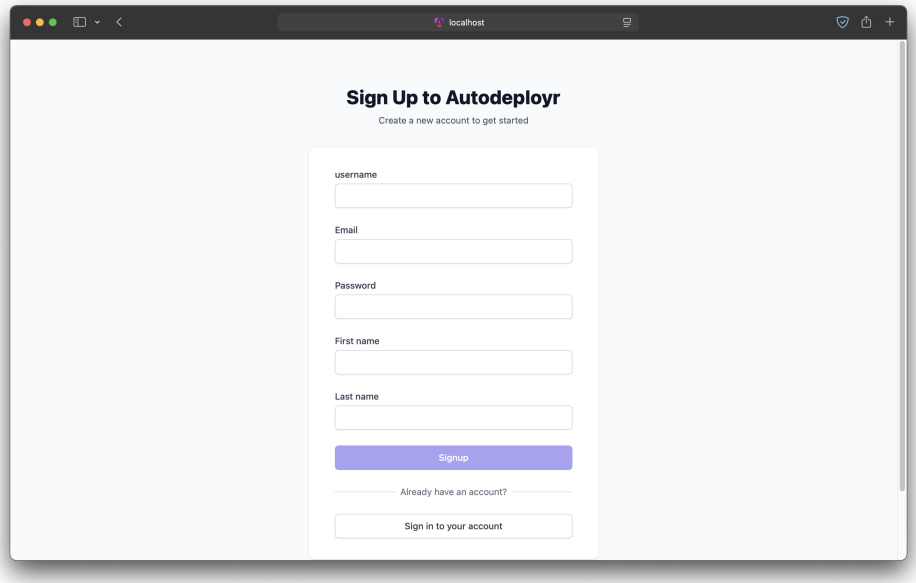

Figure 1.1: User registration.

1.3.2 General information

On the main dashboard of the interface [Figure 1.2], users can view general information about their deployed serverless functions. in ”Overview” they can see the total number of functions deployed, the total number of invocations from all of them, and the number of apps. They can also view 3 of their recently deployed applications, and 5 of their most invoked functions.

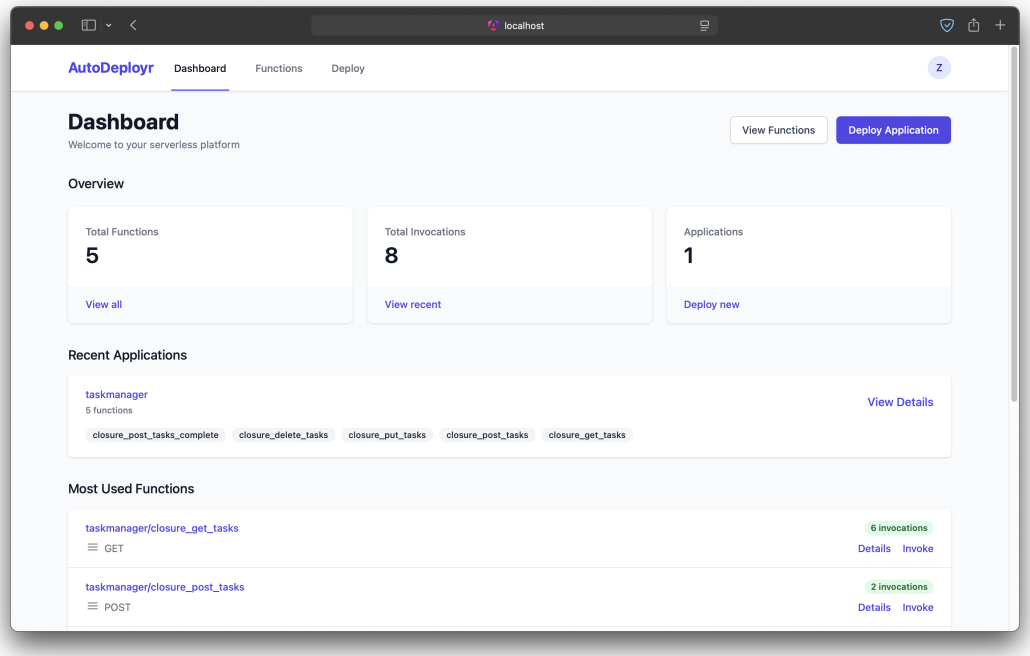

Figure 1.2: Main dashboard of AutoDeployr.

1.3.3 Application deployment

The platform supports four distinct deployment methods: ZIP file upload, direct function creation, GitHub integration and AI generation [Figure 1.3]. All of those support 3 programming languages/frameworks: Python-Flask, Java-Spring Boot and Php-Laravel.

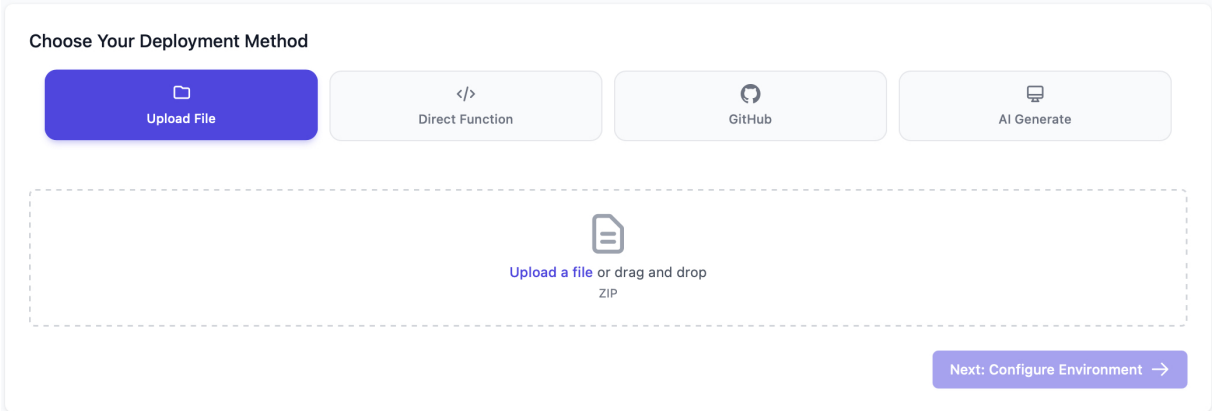

Figure 1.3: Deployment methods.

The first one represents the most straightforward method of deployment. Users prepare their application source code into a standard ZIP file and access the deploy-

ment interface through the ”Deploy application” button. The file upload area supports both drag and drop and traditional file browser selection. During upload, users can optionally set a custom app name or the ZIP file name will be set instead, configure environment variables [Figure 1.5] that their application needs and set function privacy settings. After initializing deployment, the user waits for ”Deployment successful” notification.

The direct function creation gives users the option to prototype a single function by writing only the route code or a whole app fast. Users select their preferred programming language from the supported options, then write or paste the code directly into the integrated web editor. The interface also has the option to specify the app name for an existing app or a new one. Environment variables can be set in the next step, like for the ZIP upload method.

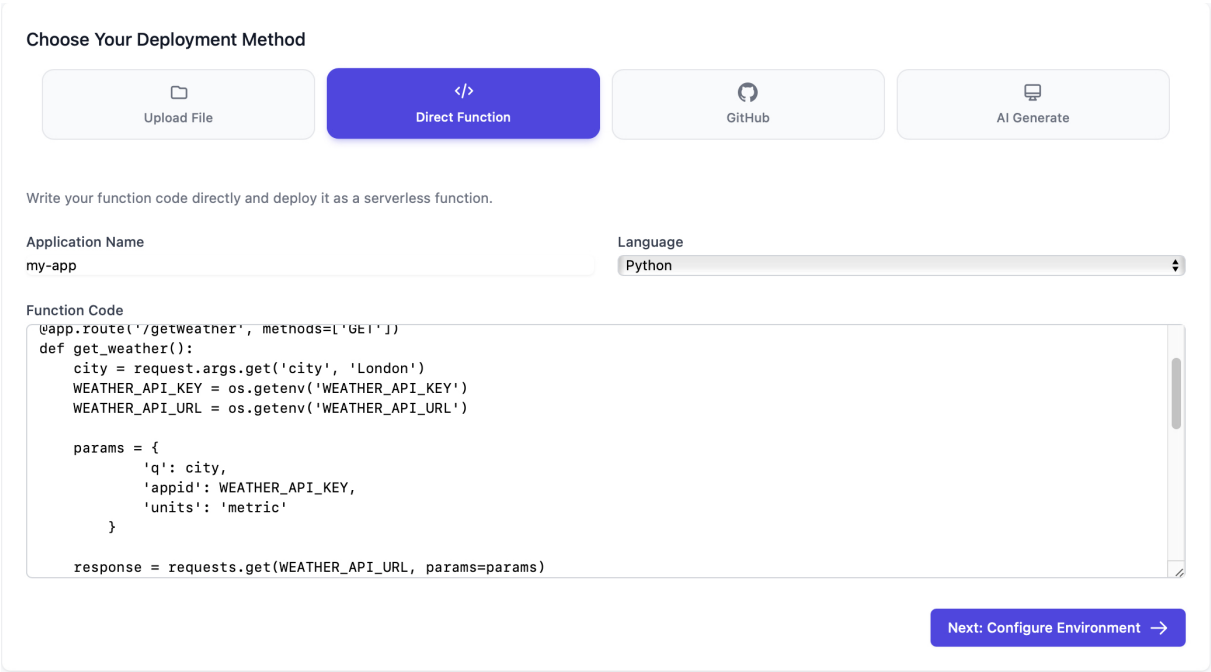

Figure 1.4: Direct function deployment.

GitHub integration makes it easier to deploy code hosted on the platform and works by providing the URL of either public or private repositories, selecting the appropriate branch, and configuring access credentials when working with private repositories. The platform automatically clones the specified branch of the repository before starting the deployment process.

The AI generation method lets users describe the desired functionality in natural

language, and the AI service processes those descriptions and generates working code in the user's desired programming language and framework. Users can review and modify the generated code before deployment. This enables even faster prototyping of new functionality with minimal coding effort.

During deployment, users can chose to set a function as private with a simple checkbox [Figure 1.5] that will generate an API key which will be used to access the function. This works for all methods of deployment and requires no additional steps or knowledge.

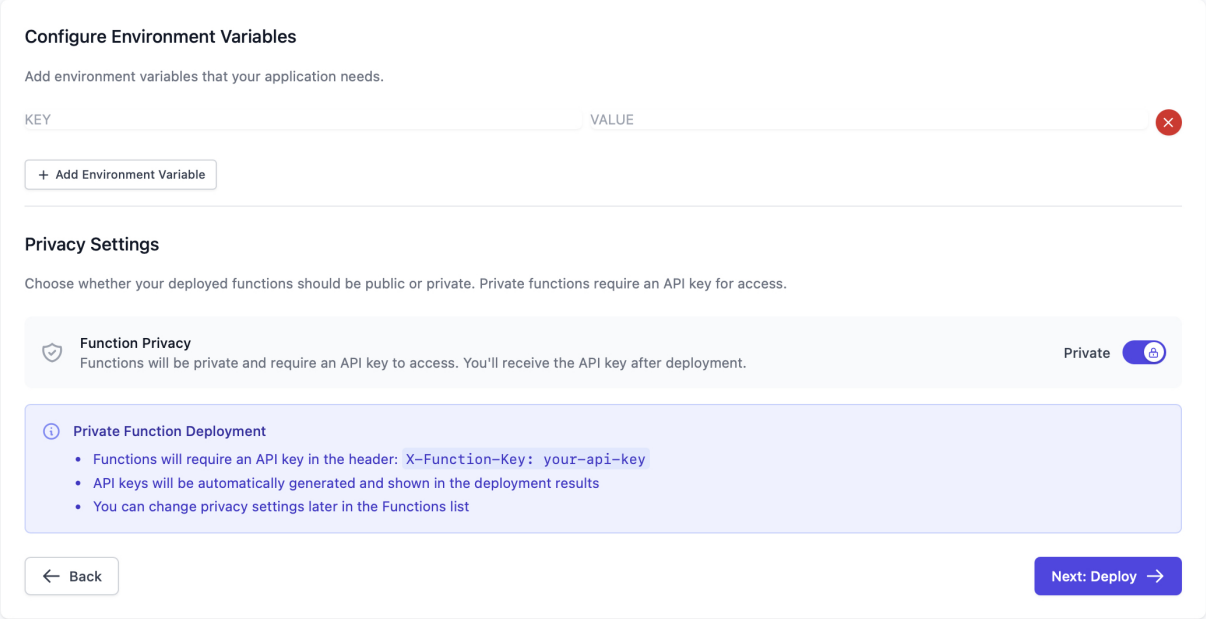

Figure 1.5: Environment variables and privacy settings.

1.3.4 Function management

After successful deployment, users can access their functions through the "Functions" tab in the navigation bar. The list displays all deployed applications with their functions and minimal metadata including the URL where a function can be accessed from, the HTTP method it uses, and the number of invocations, the language and framework the function was written in and privacy status [Figure 1.6]. Each one provides the options to test them directly from the web interface through the "Invoke" button enabling immediate testing. The performance monitoring provides more detailed information about a function, including successful and failed invocations, success rate, average, minimum and maximum execution times and the last time and date

it was invoked.

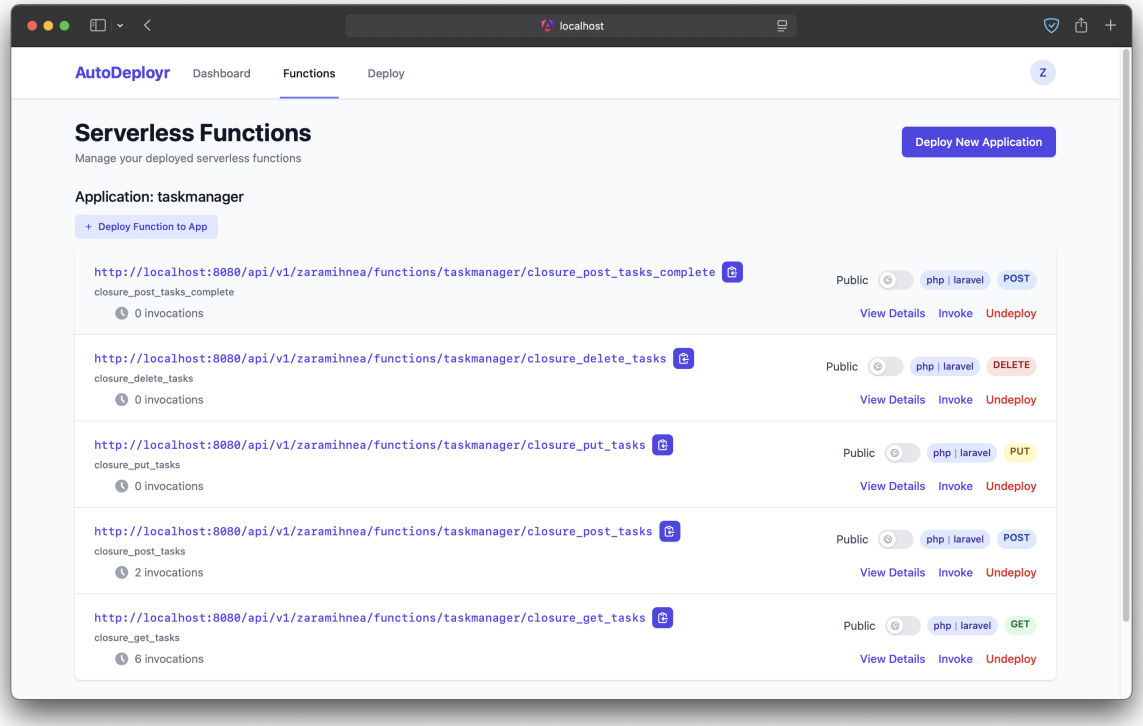

Figure 1.6: Function management.

Users can toggle each function between public and private with the simple switch. Private functions display their API keys, blurred by default, and generation timestamp [Figure 1.7]. For each applications users can add another function by pressing the ”Deploy function to app” button or switching to the ”Deploy” tab, selecting ”Direct deployment” and inserting the app name they want to add a new function to.

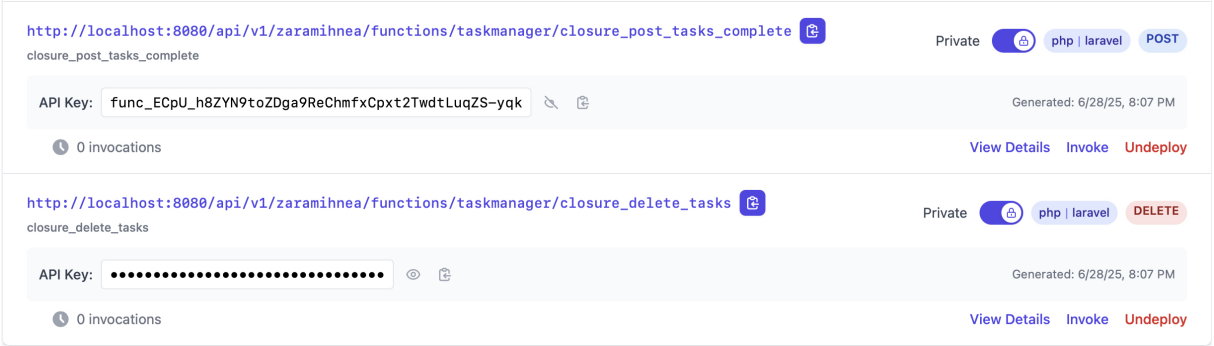

Figure 1.7: Private functions.

1.4 Similar solutions

While serverless computing matured significantly since AWS Lambda was introduced in 2014, the deployment experience from all major providers shows complexity that contradicts the paradigm's promised simplicity.

1.4.1 Existing serverless platforms

Serverless market is dominated by three providers, each with different deployment methods that create friction for developers. Despite their similar execution models, the deployment experience is very different in terms of complexity.

AWS Lambda offers the most modern set of features but also has the worst learning curve for HTTP endpoint deployment. For public endpoints, it now offers Function URLs that provide immediate HTTP access with almost no configuration, basically what serverless promised. Authenticated endpoints on the other hand require configuring API Gateway as a different service and setting up IAM roles [10, 11]. Terraform deployments show this complexity, as they require multiple resources such as aws_api_gateway_rest_api, aws_api_gateway_resource, aws_api_gateway_method and aws_api_gateway_integration just to expose a function endpoint [13, 14]. Even some experienced developers find this hard mostly due to subtle configuration requirements like integration_http_method for Lambda integrations must be POST regardless of the actual HTTP method the API exposes [15].

Google Cloud Functions tries to make things simpler by automatically generating URLs for HTTP triggered functions, so eliminating the need to create separate API services like AWS's API Gateway, but they are private by default. Developers must use `--allow-unauthenticated` flag when deploying to create public endpoints. When authentication is needed, they need to configure IAM roles. Deployment failures are very frequent though, due to dependency management issues, developers reporting errors like "Cannot find module" even with a correct package.json configuration [16]. Build errors are also common, mostly with native and transitive dependency resolutions [17, 18].

Azure Functions complicates deployment by forcing developers to chose between different hosting models before they can deploy anything. They must chose between Consumption plans, Premium plans and dedicated App Service plans [19, 20], each with some limitations and pricing models. Developers need to understand Azure's

hosting ecosystem first and then make decisions about scaling, pricing and resource allocation before deploying a function, things that other platforms do automatically. Authentication complexity then is different based on the hosting plan. It requires configuring client secrets, and authorization levels even for simple endpoints.

1.4.2 Current deployment approaches

Deployment solutions evolved into three categories, each with limitations that make it more complex.

Platform native tools like AWS SAM, Google Cloud Functions CLI and Azure Functions Core Tools require knowing each provider's specific way of doing things. The first needs YAML files that reference CloudFormations and IAM policies [10]. You are locked into the platform from the start.

Third party frameworks like Serverless Framework and Terraform say they work with multiple cloud providers but they are still complex. Basic setups need hundreds of lines of configuration [13, 14] and DevOps knowledge that most developers don't have.

Container approaches package everything in Docker containers, AWS Lambda supporting up to 10 GB images [21, 22]. But developers still need to configure networking and scaling.

1.4.3 Limitations of current solutions

Current platforms treat deployment as a secondary problem, optimizing execution while leaving deployment as a manual process. Academic research shows that serverless applications made up of multiple functions create more complex architectures and are very difficult to manage, integration testing becoming challenging [23].

Despite years since the serverless paradigm appeared, current serverless migration approaches need extensive manual decomposition of monolithic applications [24, 25]. This manual process shows the challenges of serverless function composition and state management that the industry still has to resolve [23].

Platform specific knowledge creates immediate lock in. Every AWS Lambda deployment decision, from API Gateway configuration to IAM policies, traps applications to AWS-specific services and billing models [10, 11]. These are architectural commitments that make migration to other platforms expensive and risky.

Cold start performance varies by platform. Comparative studies show AWS Lambda has cold starts under 1 second, Google Cloud Functions between 0.5 and 2 seconds and Azure Functions up to 5 seconds [26]. AWS offers ”provisioned concurrency” to keep functions warm and avoid cold starts from the start, but Google does not do this [27]. The more dependencies you include, the worse the cold starts [28, 29], which affects user experience.

This result is a serverless ecosystem that make deployment complexity worse then operational complexity that serverless computing was designed to eliminate. Analysis shows that while serverless platforms evolve to have better tools and runtimes, deployment complexity is still a big barrier in adoption [24, 25], developers spending more time configuring than writing code for applications. A study confirms that developers often run into problems with configuring environment settings and executing deployments on serverless platforms [30].

Chapter 2

System Architecture

2.1 General architecture

AutoDeployr implements clean architecture principles [31] with clear separation of concerns across five distinct layers. This makes sure we can extend and maintain the code in the future.

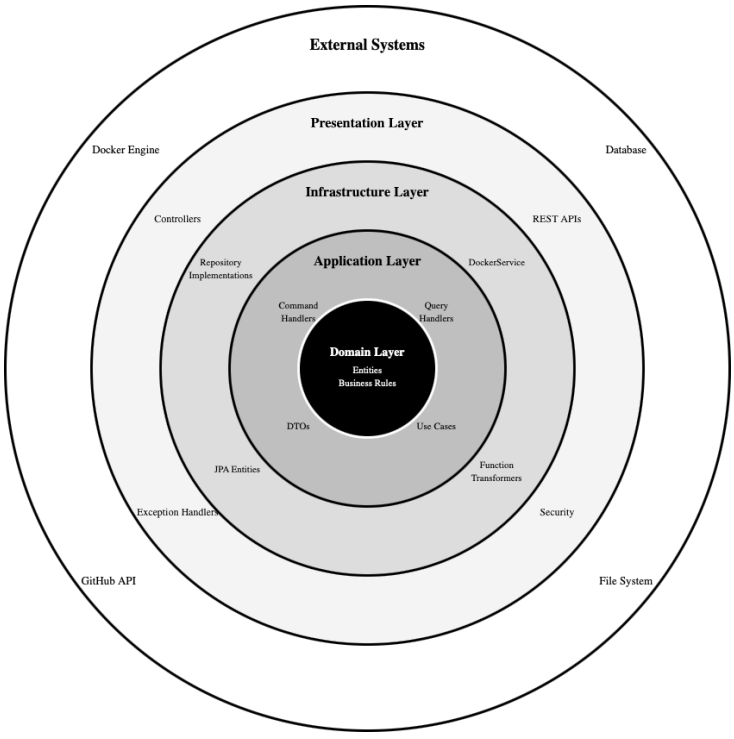

Figure 2.1: Backend clean architecture.

The Angular 19 [32] frontend manages user interaction and provides the deployment interface, communicating with the backend through RESTful APIs. The Spring Boot 3.2.2 web layer handles HTTP request processing, authentication, and re-

sponse formatting. The application layer orchestrates business workflows using Command Query Responsibility Segregation (CQRS) [33, 34] to separate modifying operations from read-only queries. The domain layer contains the entities and rules, while the infrastructure layer manages external dependencies including Docker integration, database persistence, and code analysis services.

The multi-module Maven [35] structure organizes the code into specialized modules: analyzers for code parsing, domain for core entities, application for defining the use cases, infrastructure for integrating the external services and webapi for managing the HTTP endpoints. This organization was proved essential when extending language support to Java Spring, requiring only creating a new analyzer, without modifying the existing Python Flask implementation.

2.2 Backend

2.2.1 Tech stack and configuration

The backend uses Java 21 with Spring Boot 3.4.4 for web layer functionality and data persistance.

The Maven project structure:

```
AutoDeployrBackend/
|-- analyzers/ (code analyzers)
|-- domain/ (entities and business rules)
|-- application/ (use cases, commands/queries and handlers)
|-- infrastructure/ (external services and database persistance)
`-- webapi/(REST controllers and HTTP handling)
```

Some important dependencies are Docker Java Core 3.4.2 [36] for docker container management, JavaParser 3.25.1 for Java source code analysis, Eclipse JGit 6.6.1 [37] for GitHub integration, JJWT 0.11.5 [38] for JWT token handling, BCrypt [39] for password security and PostgreSQL JDBC [40] driver for the database connection.

2.2.2 Application Layer

The application layer implements CQRS to handle deployment operations and data retrieval.

The commands are data modifying operations like DeployApplicationCommand, UndeployFunctionCommand. These handle complex errors like partial deployments, dependency resolution errors or container build errors.

Queries are optimized for reading data. Some examples are GetUserFunctions-Query which retrieves a user's deployed functions with pagination and sorting and GetFunctionMetricsQuery which gets performance statistics for a certain function.

2.2.3 Code analysis system

The code analysis system is the main innovation of AutoDeployr. It uses Abstract Syntax Trees to understand application code instead of treating application like boxes that need to be packaged up. The Python Flask analyzer breaks down the analysis into eight specialized components, each focused on understanding specific parts of an application. When an app is deployed, FlaskAppVisitor first scans through the code to find where the Flask app is defined, then FlaskRouteVisitor extracts all route decorators ("@app.route", "@blueprint.route") along with their HTTP methods and paths, FunctionDefinitionVisitor extracts the complete source code for each function, including decorators while FunctionCallVisitor builds method call graphs to map the dependencies between functions. The analyzer walks through the function calls graph recursively to find every helper function and utility method that each route needs. ImportUsageAnalyzer then looks for which imports are referenced in each route's AST so functions only include necessary dependencies. EnvVarDetector looks in the code for patters like "os.getenv()" and "os.environ[]" and Database Detector identifies database library usage through method call pattern matching. The output is a JSON structure containing extractable serverless functions with their complete metadata: [Figure 2.2]

The Java Spring Boot analyzer uses JavaParser 3.25.1 to parse Java Spring Boot applications and works with three main visitor classes that work through the code's AST to extract different information. ControllerVisitor finds Spring controllers by looking for "@RestController" and "@Controller" annotations, and for each controller it extracts endpoint mappings from annotations like "@RequestMapping", "@GetMapping" or "@PostMapping", saving the HTTP paths, methods and parameter details. DependencyVisitor creates function call graphs by tracking which methods call which other methods. It resolves method calls across classes and packages, creating a complete dependency tree which includes service methods, repository calls, and utility

functions. EndpointVisitor scans for configuration requirements by finding "@Value" annotations and extracting environment variable references from Spring's property placeholder syntax like `${DATABASE_URL}` and `${API_KEY:default}`. JavaParser's symbol solver uses ReflectionTypeSolver and JavaParserTypeSolver to understand what types and methods are being used throughout the code, which lets it accurately track dependencies even when Spring is doing dependency injection. The output, like the python analyzer, is a JSON structure with extractable serverless functions, their dependencies and configuration requirements.

The PHP Laravel analyzer uses nikic/php-parser 4.15 to parse Laravel applications. The framework has a different paradigm where routes are defined separately from their handlers. RouteVisitor analyzes the route files (routes/web.php, routes/api.php) and extracts the route definitions like `Route::get('/users',[UserController::class, 'store'])`. ControllerVisitor scans "app/HTTP/Controllers/" to extract the route handlers. The main problem is cross referencing the route definitions to their handlers. This is solved during the function extraction, when the analyzer matches the names of the routes and controllers. This method is simpler then Java Spring's JavaParser symbol solver since it can break if developers rename the classes. The output is the same as Flask and Spring analyzers, but includes PHP only metadata like class names, which enable the reconstruction of the applications as serverless functions.

```
{
  "language": "python",
  "framework": "flask",
  "app_name": "app",
  "functions": [
    {
      "name": "health",
      "path": "/health",
      "methods": [
        "GET"
      ],
      "source": "@app.route('/health')\ndef health():\n    \"\"\"Health check endpoint.\"\"\"\n    env = os.environ.get('ENV', app.config['ENV'])\n    status = HealthStatus('healthy', env)\n    return jsonify(status.to_dict())",
      "app_name": "app",
      "dependencies": [],
      "dependency_sources": {},
      "imports": [
        {
          "module": "flask.jsonify",
          "alias": "jsonify"
        },
        {
          "module": "os",
          "alias": "os"
        },
        {
          "module": "flask.Flask",
          "alias": "Flask"
        },
        {
          "module": "flask.request",
          "alias": "request"
        }
      ],
      "env_vars": [
        "PORT",
        "FLASK_ENV",
        "ENV"
      ],
      "file_path": "test.py",
      "line_number": 23,
      "requires_db": false
    }
  ]
}
```


Figure 2.2: Output of Python analyzer.

2.2.4 Deployment

AutoDeployr transforms applications into serverless functions and DeployApplicationCommandHandler manages the flow from source code to the docker containers. This flow describes the deployment of a Python Flask application.

First, the system scans the input directory for python files, and identifies Flask instances. FlaskAppVisitor locates the declaration or blueprint definition.

The analysis layer generates Abstract Syntax Trees for each python file and starts 5 visitors in parallel. ImportCollector extracts all imports, FunctionDefinitionVisitor saves function definitions and their source code, separating route handlers from util-

ity functions, FlaskRouteVisitor looks for route decorators and their endpoint paths, HTTP methods, FUnctionCallVisitor maps function call relationships for dependencies and EnvVarDetector scans for environment variables. Every route is then analyzed by ImportusageAnalyzer to check which imports are actually used and DatabaseDetector looks for database code usage. The output is made up by ServerlessFunction object that include function metadata, route paths, HTTP methods, source code, dependencies, imports used by that function and environment variables.

The code generation layer transforms this response objects into deployable artifacts with PythonFunctionTransformer. The generated files are main.py with Flask entry points, function_wrapper.py for the execution and HTTP processing, requirements.txt for the filtered dependencies and the dockerfile for building the python app and to inject the environment variables.

The system then creates isolated containers for each serverless function using the generated dockerfiles. [Figure 2.3]

2.2.5 Docker containers

The Docker integration manages the container life cycle for the execution of serverless functions. DockerService builds language-specific images using dockerfiles, handles the injection of environment variables and executes functions in containers with a 90 seconds execution timeout. No explicit memory limits are set, so the containers can use the system defaults.

Docker image naming respects the structure `autodeployr-[userID]-[appName]-[functionName]_[HTTPmethod]` to isolate and prevent conflicts between, for example, functions with the same name but owned by different users, or functions with the same name but different HTTP methods. The system automatically shutdowns containers after idle timeouts and cleans unused images.

The execution works by transforming HTTP requests to JSON, creating the containers from the docker images, passing the data to them through command line arguments for Python and Java and environment variables for PHP executing the functions directly through CLI commands (php function.php, java -jar function.jar, python function_wrapper.py), getting the response through stdout and immediately closing the container.

2.2.6 AI integration

The AI code generation system integrates OLLAMA locally at `http://localhost:11434` and uses the CodeLlama [41] model. The C# service (CodegenerationService) uses the OLLAMA REST API to generate framework specific code based on natural language.

This starts by constructing prompts using language specific system prompts saved as template files saved at `Templates/Language/system-prompt.txt` directory which are combined with the user request, and sending them to OLLAMA's generation endpoint. The request structure is:

```
var ollamaRequest = new Dictionary<string, object>
{
    ["model"] = "codellama",
    ["prompt"] = $"{systemPrompt}\n\n{prompt}"
};
```

The output is returned in JSON which is cleaned by the service of formatting like ```` ```language``` ```` and sends it back to the frontend.

2.2.7 Function security system

Autodeployr implements function security through the CQRS pattern with ToggleFunctionSecurityCommand. This system uses the Function entity's `isPrivate`, `apiKey` and `apiKeyGeneratedAt`, which enables access control for each function. This works through `/api/v1/functions/functionID/security` api endpoint. When a user toggles functions to private, the command handler generates API keys using Java's SecureRandom class and updates the metadata automatically. Function invocation validates the `X-Function-Key` header for the private functions

2.3 Frontend

2.3.1 Tech stack and configuration

The frontend is built using Angular 19. The application structure organizes features into modules:

```
AutoDeployrFrontend/src/app/
|-- components/
|    |-- auth/ (login, signup)
|    |-- dashboard/ (overview, metrics)
|    |-- deployment/ (multi-method deployment)
|    |-- functions/ (management, invocation)
|    |-- layout/ (navigation, structure)
|-- services/ (HTTP clients)
|-- guards/ (authentication protection)
|-- interceptors/ (token injection)
`-- models/ (data models)
```

2.3.2 Dashboard interface

The dashboard provides an overview of the deployed serverless functions [Figure 2.4]. It shows key metrics like number of deployed functions, total number of invocations and the number of applications. The interface also shows recently deployed applications and the top most used functions with basic information.

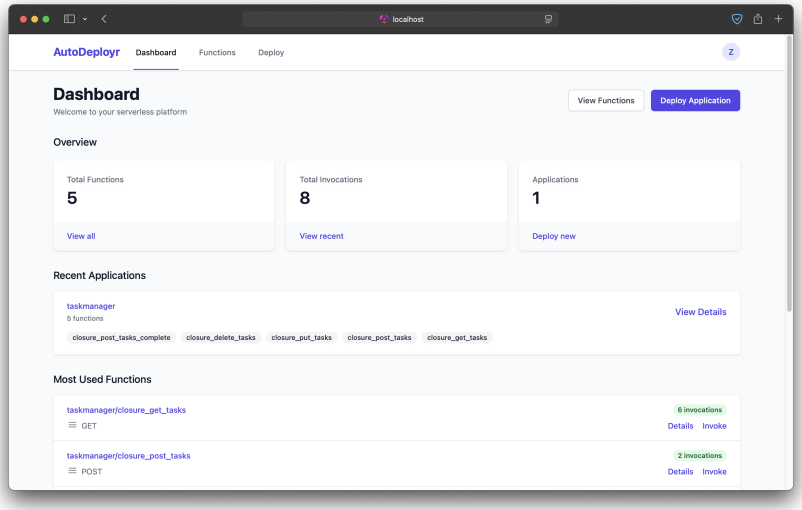

Figure 2.4: Dashboard.

2.3.3 Function management interface

The functions page shows a detailed breakdown of each application [Figure 2.5]. The functions have the full URLs displayed, with basic information like number of

invocations, HTTP method, programming language and framework. Each of them have the privacy toggle, that, when switched, the API key will appear. Both the URL and API key can be copied with the copy button. Users can test each serverless function using the invoke button, and view metrics using the view details button. The undeploy button lets users delete functions.

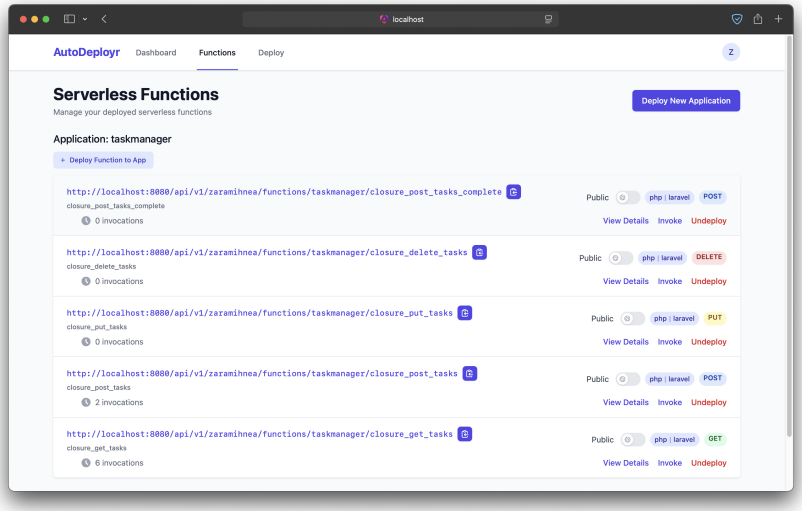

Figure 2.5: Function management interface.

2.3.4 Deployment interface

The deployment interface provides 4 methods of automatic deployment [Figure 2.6]. File upload deployment lets users drag and drop ZIP files, direct code deployment which offers a basic code editor (textarea) for writing functions directly in the browser, GitHub integration enables repository deployment with branch selection, and AI deployment which generates code with the OLLAMA service. Each deployment method has form validation. The interface provides real time feedback during the deployment process like notifications about partially or totally failed deployments.

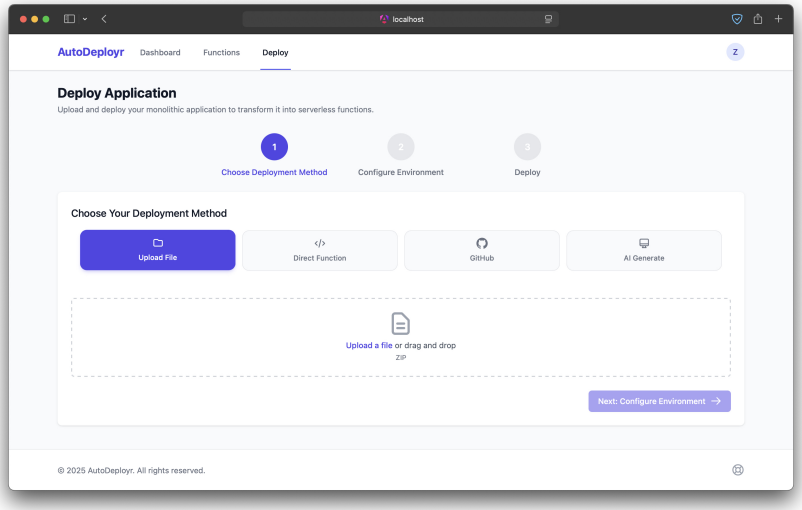

Figure 2.6: Deployment interface.

2.3.5 Services and HTTP communication

The frontend uses Angular's HttpClient for all API communications. Interceptors automatically add authentication tokens to requests and manage the loading state.

The AuthService manages the JWT token which means login, validation, renewal and browser storage. Tokens are stored in localStorage and have automatic cleanup.

2.4 Data storage

The PostgreSQL database uses Spring Data JPA with Hybernate and connection pooling

```
spring.datasource.url=jdbc:postgresql://localhost:5433/autodeployr
spring.datasource.username=postgres
spring.jpa.hibernate.ddl-auto=update
```

2.4.1 Function data

The database schema revolves around the functions table as the primary entity. This stores the essential metadata like ID, source code, user ownership and programming language.

`functions: created_at, updated_at, app_name, id, name, path, user_id, db_imports_json, imports_json, source, framework, language,`

`api_key, api_key_generated_at, is_private,`

Other tables support the function entity:

`function_entity_methods: function_entity_id, method`

`function_entity_dependencies: function_entity_id, dependency`

`function_entity_env_vars: function_entity_id, env_var`

This approach supports storage of metadata without requiring modifying the schema when adding support for new programming languages.

2.4.2 User and authentication data

The users table stores authentication credentials and account information. Password storage uses BCrypt hashing with SpringSecurity's default configuration.

`users: id, username, email, password, first_name, last_name, created_at, updated_at, active`

`user_roles: user_id, role`

User account support role based access through the user_roles table, separating regular users from administrators. The administrator interface is not implemented yet.

2.4.3 Environment variables

Because environment variables need to be injected into the docker containers, they are encrypted using AES encryption with 256 bit keys. During container startup, values are automatically decrypted and injected as plain text environment variables, which allows the code to access data like API keys and database URLs.

2.4.4 Function Metrics

The function_metrics table contains performance statistics for display in the function-detail component in the frontend. The saved information includes invocation counts, execution timing and success/failure counts. Performance data is collected when a function is executed.

2.5 Security aspects

2.5.1 JWT authentication

Authentication implements JSON Web Tokens using JJWT library. Token validation happens through Spring Security filters that extract bearer tokens from authorization headers inside HTTP requests and validate expiration times. JWT tokens are set to expire in 24 hours.

2.5.2 Password security

The passwords are hashed using BCrypt algorithm with Spring Security's PasswordEncoder interface.

```
@Bean public PasswordEncoder passwordEncoder() {
 return new BCryptPasswordEncoder();
}
```

User registration has validation for password complexity, email format verification and unique username constraint. The authentication process compares submitted passwords against stored BCrypt hashes, without exposing credentials in plaintext.

2.5.3 API security

API endpoints have extensive input validation using Jakarta Bean Validation annotations and occurs at the controller-level, with standard error responses for invalid input.

CORS configuration permits access only from approved frontend domains for most endpoints, but the `/api/v1/[username]/functions/[appName]/[functionName]` invocation endpoint allows access from any domain, with any HTTP method.

Function security is independent from platform authentication. Private functions need API keys provided through "X-Function-Key" header. This gives users the ability to have selective security based on user prefrences.

File upload security has type validation (ZIP only) and size limitations.

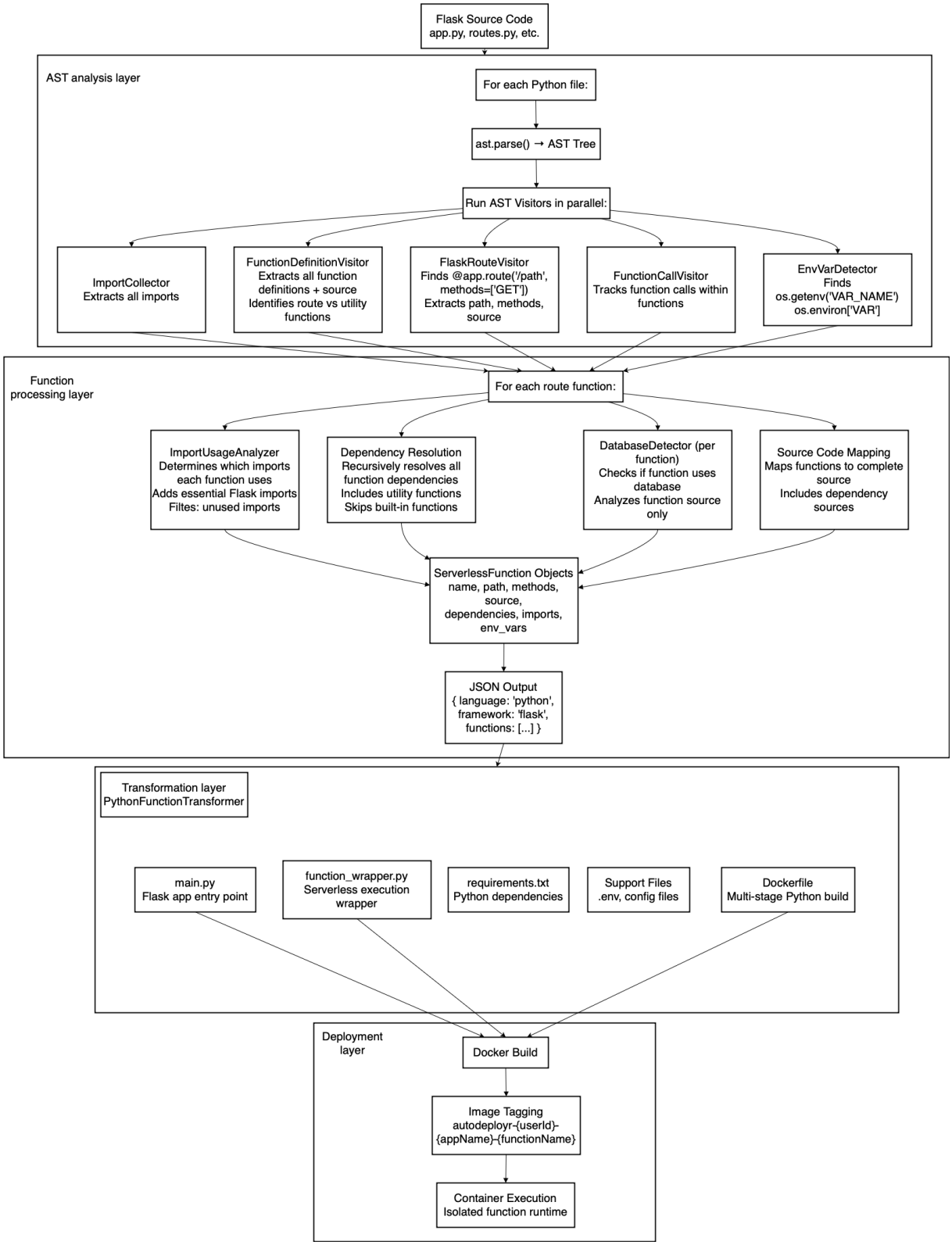

Figure 2.3: Deployment workflow of a Flask application.

Chapter 3

Use Cases

AutoDeployr demonstrates practical value for automated serverless deployment, and addresses several real world scenarios. These use cases appeared from analyzing developer frustration with existing platforms.

3.1 Backend API development for Mobile Applications

A developer needs to create Rest API endpoints for a food delivery mobile app, the main features being user registration, restaurant listings, order management and payment processing. He has experience with Flask, but no knowledge with cloud deployment methods.

With AutoDeployr, the developer writes a standard Flask app and uploads the ZIP file [Figure 3.1]. Autodeployr automatically identifies the 4 distinct endpoints and extracts dependencies like requests and stripe, and generates individual serverless functions. The deployment produces immediately accessible endpoints like "/dev_user /functions/food-app/register-user", "/dev_user/functions/food-app/list-restaurants", "/dev_user/functions/food-app/create-order" and "/dev_user/functions/food-app /register-payment". The developer can test each function individually through the function invocation interface to verify that user registration works correctly, restaurant listings return proper data, orders are accurate and payments are handled well.

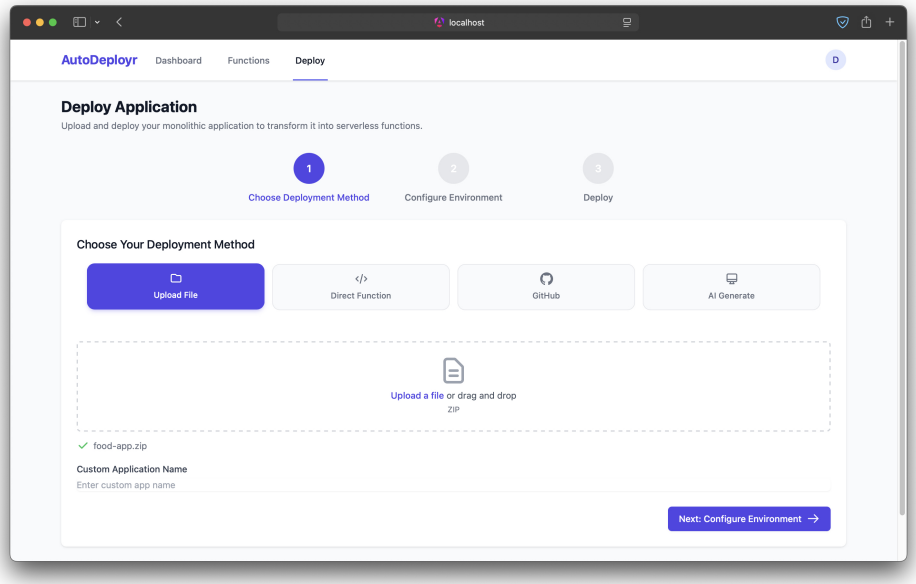

Figure 3.1: ZIP deployment method.

After the deployment process, the developer can configure endpoint privacy through the platform interface, setting sensitive endpoints like payment processing to private, requiring the API key to access them [Figure 3.2], and keeps public endpoints like restaurant listings accessible by anyone. This security configuration is applied instantly, without any infrastructure setup.

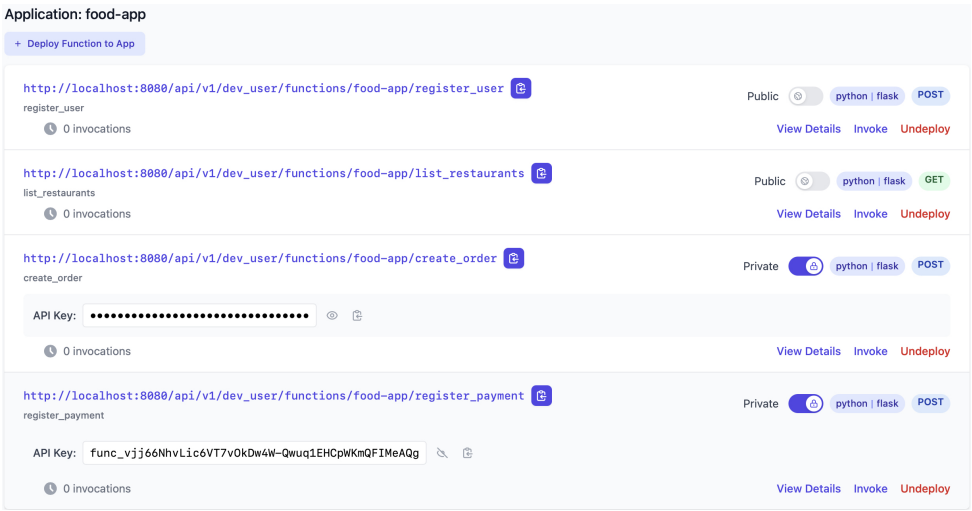

Figure 3.2: Private and public deployed functions.

What would require a few days of AWS configuration and debugging is completed in 5 minutes. The developer can now test the API endpoints directly from the AutoDeployr interface and work on integrating them in the fronted mobile application.

3.2 Rapid prototype development

A startup needs to validate a business idea that requires integration with multiple APIs: payment processing, data analytics and data retrieval from a database. They need a working prototype within days to show to potential investors, but lack the time and serverless knowledge to successfully implement an MVP.

The founder uses AutoDeployr's AI generation feature to create the required endpoints using natural language like "Create an app with 3 endpoints: a payment processing endpoint that accepts credit card information and uses Stripe for charging the customer with Stripe API key as environment variable, one endpoint that logs transaction data for analytics, and another endpoint with HTTP method POST that retrieves all data from the Supabase postgreSQL database about a transaction using the transaction ID." The AI service generates code with proper input validation and external API integration. After reviewing the code, the founder inserts the Stripe API key and database connection URL and password as environment variables, then deploys directly from the web interface [Figure 3.3].

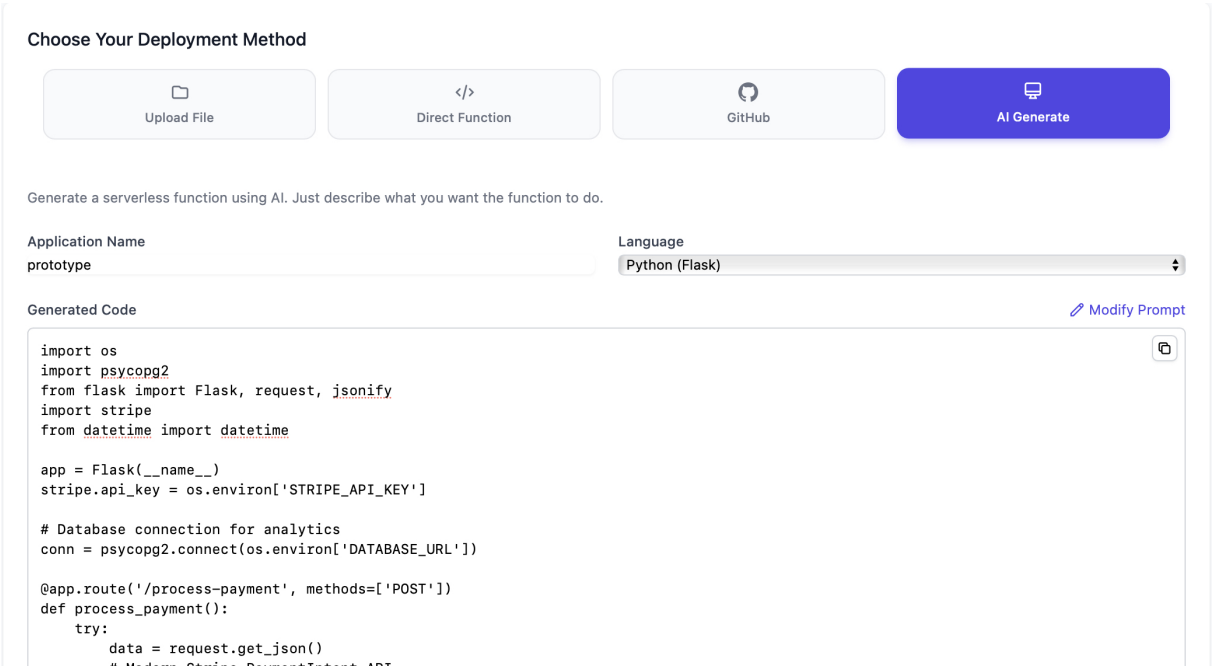

Figure 3.3: AI deployment method interface.

3.3 Multi framework development team collaboration

A software developer company has teams specialized in different frameworks, one in Flask, one in Spring Boot and one in Laravel. They need to create an e-commerce platform together.

Because AutoDeployr supports multiple frameworks, every team can develop in what language and framework they are best at. The Python Flask team can create the endpoints to list the products, the one that knows Spring Boot can create the payment processing and the PHP Laravel team can contribute with the user management functionalities. Each team uploads their projects as ZIP files and the platform deploys them as serverless functions that can be accessed from API endpoints [Figure 3.4]. This eliminates the need to manage different deployment necessities for different frameworks.

Figure 3.4: Deployed app with functions written in different programming languages.

3.4 Healthcare portal

A healthcare company has 3 separate applications developed by different contractors, one for patient scheduling one for medical records and one for prescriptions. The company needs them integrated into a single system.

Because each application uses different database connections and API keys for external services, AutoDeployr is the perfect choice for deploying the 3 applications. The IT team uploads each application as a ZIP file. AutoDeployr's analyzer automatically identifies 8 scheduling functions, 12 medical records functions, and 6 prescription functions. The system detects that each app uses different databases and external APIs, PostgreSQL and Twilio for scheduling, MongoDB and insurance APIs for

records, MySQL and pharmacy networks for prescriptions [Figure 3.5].

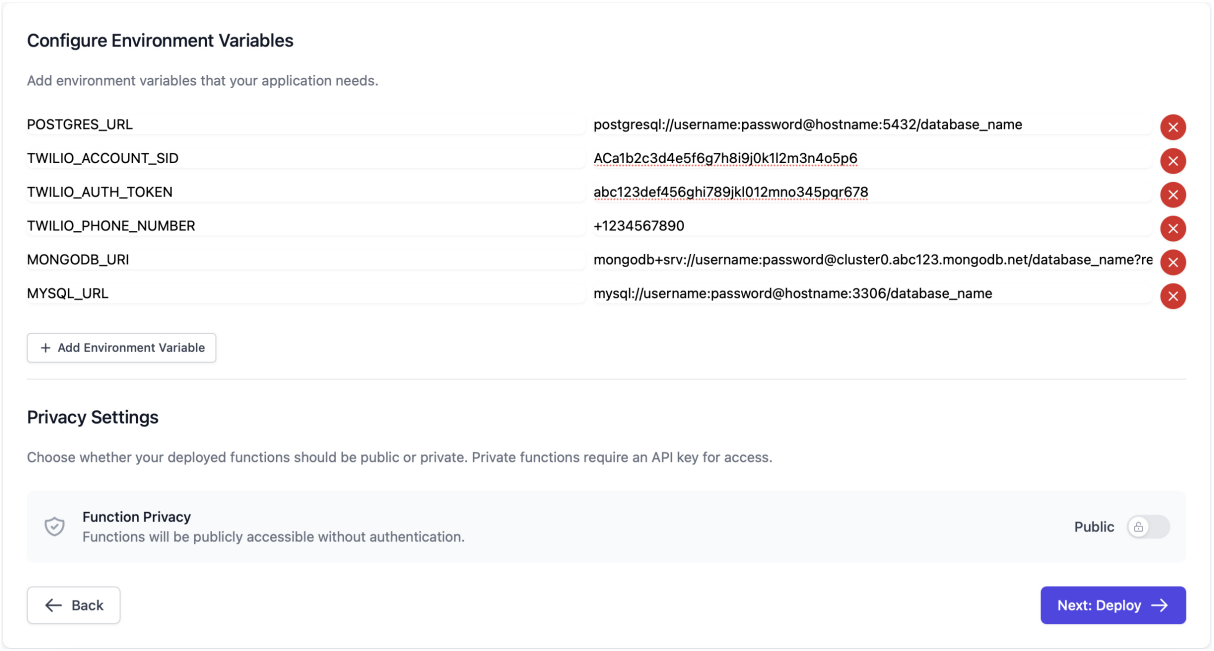

Figure 3.5: Deployment interface showing multiple environment variables.

AutoDeployr extracts all the environment variables and deploys the functions with endpoints like `/healthcorp/functions/scheduling/book-appointment`, `/healthcorp/functions/records/get-patient-history`, and `/healthcorp/functions/prescriptions/lookup-medication`. The healthcare company can now build a unified patient portal that seamlessly calls these individual functions. When a patient books an appointment, the system automatically retrieves their medical history and prescriptions, creating an integrated experience from three originally separate applications.

Conclusions and future improvements

AutoDeployr successfully demonstrates that automatic serverless deployment is technically feasible and valuable. The platform eliminates the manual decomposition requirement that is a big problem in serverless adoption and provides a working solution for automatically deploying applications into containerized serverless functions. The platform solves problems that make automatic deployment difficult. The code analysis can take a Flask app with multiple routes and figure out which dependencies each endpoint needs, then build separate lightweight containers instead of bloated ones. It handles tracking function calls between files and detecting when code uses environment variables or database connections. The AI integration with OLLAMA lets users describe what they want and get working code.

The deployment automation works across the supported languages and frameworks well, analyzing, transforming and deploying applications without requiring any past knowledge or configuration. Function invocation through the HTTP endpoints give users immediate access to deployed functionality with built in monitoring. From a user experience perspective, the platform delivers on its promise that developers can automatically deploy backend applications fast and receive working endpoints.

While AutoDeployr proves the concept of automatic serverless deployment, there are several problems that prevent immediate adoption. Scalability constraints is the biggest one, as the platform runs on a single server infrastructure without load balancing or database scaling and the user isolation also relies on naming conventions. These decisions were enough for research validation, but the platform requires changes for production deployment. The security implementation is enough for demonstration purposes, but the platform would benefit for security scanning of user-submitted code. The container isolation prevents users from accessing each other's functions, but harmful code could impact platform stability.

Container management has no memory limits and uses a fixed 90 second execu-

tion timeout, and does not optimize for different workloads. In the future, it should be implemented adaptive memory allocation based on actual memory usage patterns so it would help with memory starvation of other functions. Developer experience is also limited by the platform not having the function code visible and the ability to edit it, which would help with bug fixing without having to redeploy the whole app. Language support is another problem that will be fixed in the future, mainly the fact that the platform only has support for some popular web frameworks. Adding a new language or frameworks means implementing new analyzers and transformation logic.

AutoDeployr proves that automatic serverless deployment works, but also that this approach brings its own challenges . This approach eliminates manual decomposition requirements and demonstrates multi-language deployment automation, but the performance limitations and scalability constraints show why existing platforms have not pursued similar strategies.

Bibliography

[1] Peter Sbarski and Sam Kroonenburg. *Serverless Architectures on AWS: With examples using AWS Lambda*. 2021.

[2] Weisong Sun, Chunrong Fang, Yun Miao, Yudu You, Mengzhe Yuan, Yuchen Chen, Quanjun Zhang, An Guo, Xiang Chen, Yang Liu, and Zhenyu Chen. Abstract syntax tree for programming language understanding and representation: How far are we? 2023.

[3] Flask Development Team. *Flask Quickstart Guide*, 2025. URL `https://flask.palletsprojects.com/en/3.0.x/quickstart/`.

[4] Python Software Foundation. *Abstract Syntax Trees*, 2025. URL `https://docs.python.org/3/library/ast.html`.

[5] Spring Team. *Spring Boot Reference Documentation*, 2025. URL `https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/`.

[6] JavaParser Team. *JavaParser Documentation*, 2025. URL `https://javaparser.org/`.

[7] Laravel Team. *Laravel Documentation*, 2025. URL `https://laravel.com/docs`.

[8] Nikita Popov. *PHP-Parser Documentation*, 2025. URL `https://github.com/nikic/PHP-Parser`.

[9] OLLAMA Team. *OLLAMA Documentation*, 2025. URL `https://ollama.com/`.

[10] Amazon Web Services. *Invoking a Lambda function using an Amazon API Gateway endpoint*, 2025. URL `https://docs.aws.amazon.com/lambda/latest/dg/services-apigateway.html`.

[11] Amazon Web Services. *Tutorial: Using Lambda with API Gateway*, 2025. URL https://docs.aws.amazon.com/lambda/latest/dg/services-apigateway-tutorial.html.

[12] Docker Inc. *Docker Overview*, 2025. URL https://docs.docker.com/get-started/overview/.

[13] DEV Community. Deploying amazon api gateway and lambda with terraform, 2025. URL https://dev.to/aws-builders/deploying-amazon-api-gateway-and-lambda-with-terraform-1i2o.

[14] Spacelift. How to create api gateway using terraform & aws lambda, 2023. URL https://spacelift.io/blog/terraform-api-gateway.

[15] LocalStack. Aws api gateway and lambda integration configuration, 2024. URL https://hashnode.localstack.cloud/the-api-gateway-lambda-tricky-integration.

[16] Stack Overflow Community. Cloud functions not downloading dependencies, 2017. URL https://stackoverflow.com/questions/46119936/cloud-functions-not-downloading-dependencies.

[17] GitHub Community. Google cloud functions - npm install fails, 2021. URL https://github.com/lovell/sharp/issues/2811.

[18] Stack Overflow Community. Google cloud functions sometimes doesn't find @google-cloud/* dependencies, 2021. URL https://stackoverflow.com/questions/65538584/google-cloud-functions-sometimes-doesnt-find-google-cloud-\dependencies-altho.

[19] Microsoft Learn. *Azure Functions Consumption plan hosting*, 2025. URL https://learn.microsoft.com/en-us/azure/azure-functions/consumption-plan.

[20] Microsoft Learn. *Azure Functions scale and hosting*, 2025. URL https://learn.microsoft.com/en-us/azure/azure-functions/functions-scale.

[21] Amazon Web Services. *Create a Lambda function using a container image*, 2025. URL https://docs.aws.amazon.com/lambda/latest/dg/images-create.html.

[22] Amazon Web Services. New for aws lambda – container image support, 2020. URL https://aws.amazon.com/blogs/aws/new-for-aws-lambda-container-image-support/.

[23] Hassan B. Hassan, Saman A. Barakat, and Qusay I. Sarhan. Survey on serverless computing. *Journal of Cloud Computing*, 10(39), 2021. URL https://journalofcloudcomputing.springeropen.com/articles/10.1186/s13677-021-00253-7.

[24] Serverless First. Migrating a monolithic saas app to serverless — a decision journal, 2019. URL https://serverlessfirst.com/serverless-migration-journal/.

[25] AWS for Engineers. Aws serverless migration: Legacy app modernization guide, 2024. URL https://awsforengineers.com/blog/aws-serverless-migration-legacy-app-modernization-guide/.

[26] Mikhail Shilkov. Comparison of cold starts in serverless functions across aws, azure, and gcp, 2021. URL https://mikhail.io/serverless/coldstarts/big3/.

[27] CloudThat. A comparative analysis of gcp and aws serverless services, 2023. URL https://www.cloudthat.com/resources/blog/a-comparative-analysis-of-gcp-and-aws-serverless-services-\google-cloud-functions-vs-aws-lambda.

[28] Ran Isenberg. Aws lambda cold start: What it is & practical ways to reduce it, 2025. URL https://www.ranthebuilder.cloud/post/is-aws-lambda-cold-start-still-an-issue-in-2024.

[29] Lumigo. What causes aws lambda cold starts & 7 ways to solve them, 2020. URL https://lumigo.io/blog/this-is-all-you-need-to-know-about-lambda-cold-starts.

[30] Jinfeng Wen, Zhenpeng Chen, Yi Liu, Yiling Lou, Yun Ma, Gang Huang, Xin Jin, and Xuanzhe Liu. An empirical study on challenges of application development in serverless computing. In *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, 2021.

[31] Robert C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. 2017.

[32] Angular Team. *Angular Documentation*, 2025. URL `https://angular.io/docs`.

[33] Martin Fowler. Cqrs - command query responsibility segregation, 2011. URL `https://www.martinfowler.com/bliki/CQRS.html`.

[34] Greg Young. Cqrs documents, 2010. URL `https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf`.

[35] Apache Software Foundation. *Maven Getting Started Guide*, 2025. URL `https://maven.apache.org/guides/getting-started/`.

[36] Docker Java Team. *Docker Java Documentation*, 2025. URL `https://github.com/docker-java/docker-java`.

[37] Eclipse Foundation. *JGit Documentation*, 2025. URL `https://www.eclipse.org/jgit/`.

[38] JJWT Team. *Java JWT Documentation*, 2025. URL `https://github.com/jwtk/jjwt`.

[39] Spring Security Team. *Spring Security Password Encoding*, 2025. URL `https://docs.spring.io/spring-security/reference/features/authentication/password-storage.html`.

[40] PostgreSQL Global Development Group. *PostgreSQL Documentation*, 2025. URL `https://www.postgresql.org/docs/`.

[41] Meta AI. Code llama, 2023. URL `https://ai.meta.com/blog/code-llama-large-language-model-coding/`.