BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

OPTIMISATION OF TESTING ENVIRONMENT ALLOCATION IN TESTING FARM SERVICE

OPTIMALIZACE ALOKACE TESTOVACÍHO PROSTŘEDÍ V SLUŽBĚ TESTING FARM

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTOR **DANIEL ŠIMKO**
AUTOR PRÁCE

SUPERVISOR **RNDr. MAREK RYCHLÝ, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

148208

Institut:	Department of Information Systems (UIFS)
Student:	**Šimko Daniel**
Programme:	Information Technology
Specialization:	Information Technology
Title:	**Optimisation of Testing Environment Allocation in Testing Farm Service**
Category:	Software Engineering
Academic year:	2022/23

Assignment:

1. Familiarize yourself with the Testing Farm project and testing system, its parts and interfaces. Focus in particular on the Artemis service, which is able to find or create and provide a virtual or physical machine suitable for running tests based on the specified parameters. Familiarize yourself with the issue of reserving or allocating a physical or virtual machine and its use and life cycle.
2. Design a method and system to speed up the allocation of the machines for the Testing Farm by preparing them in advance or by using a cache. Also design a way to monitor, manage, configure and document such system and the machines.
3. After consultation with the supervisor, implement an extension of the Artemis service using pre-allocation of machines according to the design from the previous point. Integrate the result with other services of the Testing Farm project.
4. Test the solution, evaluate and discuss the results. Publish the resulting software as open-source.

Literature:

- Testing Farm Documentation. Red Hat. 2022 [cit. 2022-09-23]. Available at https://docs.testing-farm.io/
- Prchlík, M.: Artemis. Red Hat. 2022 [cit. 2022-09-23]. Available at https://artemis6.docs.apiary.io/
- Gregg, B.: Systems performance: enterprise and the cloud. Second edition. Boston: Addison-Wesley, 2021. ISBN 978-0-13-682015-4
- De, P., Gupta, M., Soni, M., Thatte, A.: Caching VM Instances for Fast VM Provisioning: A Comparative Evaluation. In: Kaklamanis, C., Papatheodorou, T., Spirakis, P.G. (eds) Euro-Par 2012 Parallel Processing. Euro-Par 2012. Lecture Notes in Computer Science, vol 7484. Springer, Berlin, Heidelberg, 2021. ISBN 978-3-642-32820-6. https://doi.org/10.1007/978-3-642-32820-6_33

Requirements for the semestral defence:
Item 1 and 2 finished; item 3 in progress.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Rychlý Marek, RNDr., Ph.D.**
Head of Department:	Kolář Dušan, doc. Dr. Ing.
Beginning of work:	1.11.2022
Submission deadline:	10.5.2023
Approval date:	24.10.2022

Abstract

The aim of this thesis is to implement shelving, and pre-provisioning of guest virtual machines as optimizations in the VM provisioning component of a test pipeline. This work describes the process of provisioning of virtual machines in the context of the Artemis service, and the Testing Farm service environment, and modifications made to the provisioning pipeline in order to decrease the time between making a new provisioning request and receiving a fully-provisioned mechine.

Abstrakt

Cieľom tejto práce je implementácia 'poličky' a poprednej prípravy virtuálnych strojov ako optimalizácií v procese zaisťovania virtuálnych strojov pri testovaní softvéru. Táto práca popisuje proces získavania virtuálnych strojov službou Artemis v prostredí služby Testing Farm a zmeny vykonané v mechanizmoch zabezpečujúcich získavanie virtuálnych strojov tak, aby bol znížený čas medzi vytvorením požiadavku a poskytnutím plne funkčného stroja.

Keywords

Virtual machine, Provisioning, Cache, Pre-provisioning, Continuous Integration, Artemis, Python

Klíčová slova

Virtuálny stroj, Zaisťovanie virtuálnych strojov, Cache, Popredná príprava virtuálnych strojov, Priebežná integrácia, Artemis, Python

Reference

ŠIMKO, Daniel. *Optimisation of Testing Environment Allocation in Testing Farm Service.* Brno, 2023. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor RNDr. Marek Rychlý, Ph.D.

Optimisation of Testing Environment Allocation in Testing Farm Service

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of RNDr. Marek Rychlý, Ph.D. The supplementary information was provided by Miloš Prchlík. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Daniel Šimko
May 10, 2023

Acknowledgements

I want to express my gratitude towards the few people who made this thesis possible and helped me throughout the process. First, I would like to thank my consultant Miloš Prchlík for guiding me throughout this process and providing me invaluable insight into the technologies involved. Also, I would like to extend my utmost gratitute towards my supervisor RNDr. Marek Rychlý, Ph.D for providing feedback and guidance in regards to the formal aspects of the thesis.

Contents

Chapter 1

Introduction

In today's landscape of fast evolving technologies, the need for flexible and reliable resources has lead to the emergence of virtualisation technologies, such as virtual machines. One of the largest bottlenecks and a critical aspect of providing such services is the provisioning of said technologies.

The virtual machine provisioning is a key aspect of infrastructure as a cloud services. Therefore, there has been an enormous interest in increasing the ease of use and the quality of the provided services by optimising the time spent by preparing new machines. Chapter 2 provides an overview of some of the techniques used to decrease the time between a requires for a machine allocation is made until the machine becomes ready for use, as well as technologies related to the work done in this thesis.

This assignment was created in a collaboration with the Red Hat Czech s.r.o. company as they bear an interest in optimising virtual machine allocation within their testing pipelines. The relation, and the source of the interest, is explained in Chapter 3, describing the use of virtual machines for tests execution in the environment of the Testing Farm, and provides an overview for the architecture and composition of the Artemis provisioning service this thesis focuses on.

Chapter 4 delves into an in-depth analysis of the current use cases of the provisioner, discusses presented approaches and devises a design of a mechanism intended to increase the perceived responsiveness and efficiency of the service. Chapter 5 then describes the implementation specifics of the modifications to the provisioning mechanism as devised in the prior chapter.

Chapter 2

Related Work and Technologies

In recent years, virtual machine (VM) provisioning optimization has become a significant research focus due to its impact on the efficiency of cloud computing platforms. Various ways to approach this problem, as well as related avenues, have been explored. This chapter aims to review the related work focusing on various approaches and techniques employed to address the challenges associated with resource management in cloud computing platforms.

2.1 Caching

As described by Brendan Gregg [11], a cache refers to a component that stores the duplicate of or buffers a limited amount of data, such that they can be quickly retrieved, improving the performance of the system. Important metrics for evaluation of the cache efficiency are *cache hits* and *misses*, which refer to the number of times the required resources could be found, or could not be found in the cache, respectively, and notably the *hit ratio* defined as the ratio of cache hits and all attempted accesses to the cache.

Caching is also extensively explored and used outside the hardware realm, notably in the context of web resources caching and content delivery networks. The goal there is to minimize the delay and costs incurred by transferring data over long distances by being able to cache requests at a closer location. A significant challenge in the anticipation of future requests arises from the relative difficulty of prediction of future request [8].

Similar concerns are also applicable to caching in the cloud context, where likewise the topic has been extensively studied. Multiple solutions have been proposed and are being utilized as explored in the following section (2.2).

2.2 Virtual Machine Provisioning Optimization

Virtual Machine provisioning is a critical aspect of cloud computing, responsible for allocating resources to VMs and deploying them on physical hosts. Optimizing VM provisioning is essential for maximizing resource usage, minimizing costs, and reducing management overhead, which are all important factors impacting the providers and consumers of infrastructures services alike. Over time, many approaches tackling the issues associated with different parts of the provisioning of a scalable infrastructure have been described.

One of the first steps when provisioning a new VM is to look up in the image repository and copy the image template file to a compute host. These image files can be very large in size, often in the GigaByte range, and transferring such large files over the network is time consuming [9]. As Emeneker et al. [10] demonstrated, this problem can be alleviated by caching virtual images on the compute nodes themselves. Another approach, instead of focusing on efficient delivery of the images themselves to the nodes, is to efficiently deploy instances accounting for where the images are already present [12].

In addition to the overhead incurred by transferring large images, the boot process can be slow depending on the number of pre-installed components in the image. Zhu et al. [16] developed an approach, which allows to bypass the often lengthy application initialization time by leveraging the ‘suspend’ operation of a VM and creating a snapshot of an initialized system. Naive approach, as noted in the paper, consisting of loading the created memory snapshot is feasible only for systems with smaller memory, thus they developed a technique consisting of selectively loading only those memory pages that are likely to be accessed in the near future and continue loading remaining pages in parallel with the VM execution.

An approach looking into a possible solution for both of the aforementioned issues is explored by Pradipta el al. at IBM research [9]. Their proposed solution consists of predicting the expected usage of different machine instance types and used image templates, and instantiating these VMs, which would be stored in a standby mode in a cache. In essence, they create an inventory of readily deliverable VMs, which can be used to quickly serve user requests.

2.3 Resource Pooling

Pooling refers to the concept of grouping together of various resources or assets [4] in order to extract greater profit from the resources.

Resource pooling can be seen used also in the context of computing[1]. The term can be used to refer to the grouping of anything from database connections, through raw hardware resources, such as RAM or CPUs, to whole virtual machines, which will be the context commonly used throughout this work.

2.4 Message-Oriented Middleware

Edward Curry defined [7] message-oriented middleware (MOM) as referring to an infrastructure that facilitates asynchronous communication between distributed systems through the exchange of messages. This interaction model addresses many limitations found in remote procedure call (RPC) mechanisms which rely on synchronous communication and can lead to performance bottlenecks.

In a MOM-based system, clients are not required to block and wait until an operation is completed, instead, they can continue processing once a message has been sent. This allows the delivery of messages when the sender or receiver is not active or available to respond at the time of execution.

MOM systems typically employ one of, or the combination of, two primary messaging models: point-to-point and publish/subscribe. Both of these models rely on message exchange through a queue, which is typically the First-In First-Out (FIFO) queue. As the name suggests, in this type of queue, the messages are retrieved in the same order they were sent.

The **point-to-point** model allows for a direct asynchronous exchange of messages between software entities. Although usually used with only a single receiver, there is no strict

[1]VMware vSphere: `https://dzone.com/articles/resource-pooling`

restriction on the number of receivers. Rather, even with multiple connected receivers, the message is delivered only once — only to a single receiver. This model is particularly useful as it can be employed to introduce smooth, efficient load balancing into a system.

In contrast, the **publish/subscribe** model facilitates a one-to-many or many-to-many message distribution. A single (or multiple) clients can publish (send) messages, which are received by all clients subscribed to a queue — referred to as a topic.

A component of MOM solution is a **message broker**. Its role, as described by Martin Kleppmann [13], is to facilitate the routing, storage, validation and the delivery of messages to the proper destinations. Typically, they do not enforce any particular data model as a message is just a sequence of bytes with metadata.

Chapter 3

Artemis Machine Provisioning Service

This chapter describes the role of the Artemis machine provisioning service in the context of the Testing Farm, its capabilities, architecture, and the provisioning pipeline.

3.1 Testing Farm

Testing Farm [6] provides Testing System as a Service to numerous Red Hat internal, as well as external open-source projects related to Red Hat products. It enables users to submit tests to be run to a REST API, which easily integrable into other services. The test definitions are abstracted away using an open-source test metadata format, which aims to unify the way engineers, contributors, and communities are able to discover, debug, and run tests. As such it is often used as a test execution backend of other CI systems, and services.

The service is capable of executing tests defined in multiple formats. These include the Standard Test Interface (STI)[1] and a more modern Flexible Metadata Format (FMF)[2]. The tests themselves can be executed on both virtualised and bare-metal hardware environments. This allows developers to run tests on different hardware configurations as needed, without worrying about the underlying infrastructure. The test infrastructure abstraction ensures that the tests can request specific hardware requirements without being concerned about which infrastructure to use.

The testing pipeline of the service follows a well-defined sequence of steps. After a user submits a request to the service's API, a plan is generated. According to the plan(s) a number of tests in multiple environments can be executed in parallel.

Each pipeline then starts with provisioning a machine, which will be used to run the required tests. The provisioner module communicates with Artemis, a machine provisioning service. After obtaining a machine, the test environment in prepared by installing the required artifacts as required by the test specification. Afterwards, the specified tests are executed and their results along with generated artifacts are collected and stored in artifact storage. This result is then reported to the user. This pipeline is shown in the Figure 3.1.

Testing farm utilizes the TMT project[3], a tool to manage and execute tests, to handle certain steps of the pipeline. At the moment these are only the test planning stage and

[1]Standard Test Interface: https://docs.fedoraproject.org/en-US/ci/standard-test-interface/
[2]Flexible Metadata Format: https://fmf.readthedocs.io/en/stable/overview.html
[3]TMT: https://tmt.readthedocs.io/

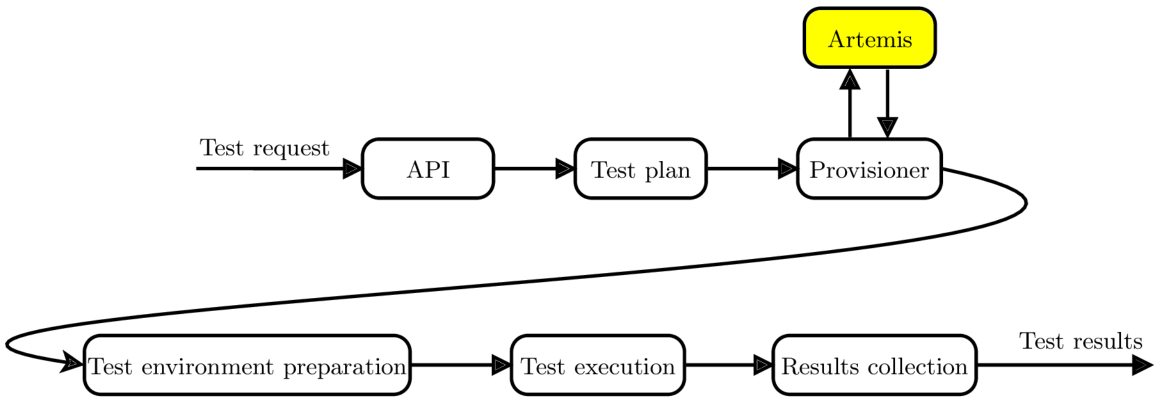

Figure 3.1: Testing Farm test pipeline

test execution, however, the plan is for TMT to handle the entire pipeline in the future. Notably, TMT can also utilize the Artemis provisioning service.

3.2 Artemis Architecture

Artemis is a machine provisioning service designed to streamline the process of allocating virtual machines for testing purposes in a pipeline implemented using the Python programming language[4]. The primary goal of Artemis is to provide a unified interface for interacting with different cloud providers, such as OpenStack or AWS, while offering fine-grained options for machine selection and configuration. This enables extensive testing on a diverse range of machine types and architectures without the need for specialized code in continuous integration (CI) or test pipelines.

Key features of Artemis include:

- Abstraction layer: Artemis abstracts the complexities of working with various cloud providers by offering a unified interface for machine provisioning. This allows users to interact with different providers seamlessly without having to worry about the specific implementation details testing-farm.gitlab.io.
- Hardware constraint specification: Users can define hardware constraints, such as a minimum amount of RAM or a specific number of CPU cores, to ensure that the provisioned machines meet their testing requirements testing-farm.gitlab.io.
- Wide range of machine types and architectures: With its fine-grained options for machine selection, Artemis enables testing on various machine types and architectures, increasing the flexibility and robustness of the testing process.
- Failure detection and fail-over mechanisms: Artemis is designed to increase the availability of machine provisioning services by implementing failure detection and fail-over mechanisms. This ensures that the service continues to function even in the presence of failures, providing a more reliable and resilient solution for users.

Artemis uses the concept of drivers and pools of resources. A specific **driver** refers to a component adapting the API of a specific cloud provider, and implementing methods

[4]Python: https://www.python.org/

required internally by Artemis to be able to communicate with the specific provider. A **pool** refers to a specific group of resources accessible by a specific user or billing account on the platform. A pool utilizes a driver to communicate with the platform, and each driver can be used by multiple pools with a different configuration and/or account.

3.2.1 Microservice Architecture

Microservice Architecture (MSA) is a software development pattern that structures applications as a collection of small, autonomous services, each implementing a single business capability within a bounded context. Microservices are independently deployable services modeled around a business domain, commonly communicating through well-defined interfaces and lightweight networks and offer various options for solving problems. They are a type of service-oriented architecture (SOA) with an emphasis on service boundaries and independent deployability. These services are technology agnostic and expose their business capabilities through network endpoints, making them a form of distributed system.

The microservices architecture offers several advantages, such as scalability and reliability. Services can be scaled independently, allowing for more efficient resource utilization and addressing bottlenecks in specific subsystems without affecting the entire application. Additionally, this architectural pattern helps enhance reliability by promoting fault isolation, ensuring that a failure in one service has minimal impact on the overall system performance. In a microservices architecture, each service is designed to be independent and responsible for a specific business function, enabling the system to recover from failures and maintain stable performance. [14]

Artemis builds on top of this pattern with the server part consisting of 4 inter-operating components:

- **API** component provides HTTP REST API, used to request, query the status of, and delete objects, such as guests (virtual machines), snapshots, metrics;
- **Dispatcher** is a component that periodically queries for new tasks submitted in the database, and upon reading them, dispatches these tasks to a broker to be processed by workers;
- **Scheduler** handles dispatch of periodically-running tasks, such as refreshing pool metrics or checking the worker availability and stability;
- **Worker** receives tasks from the broker, and executes the operations, such as guest provisioning, release, snapshot operations, etc.

A more detailed description of these components is provided in 3.5

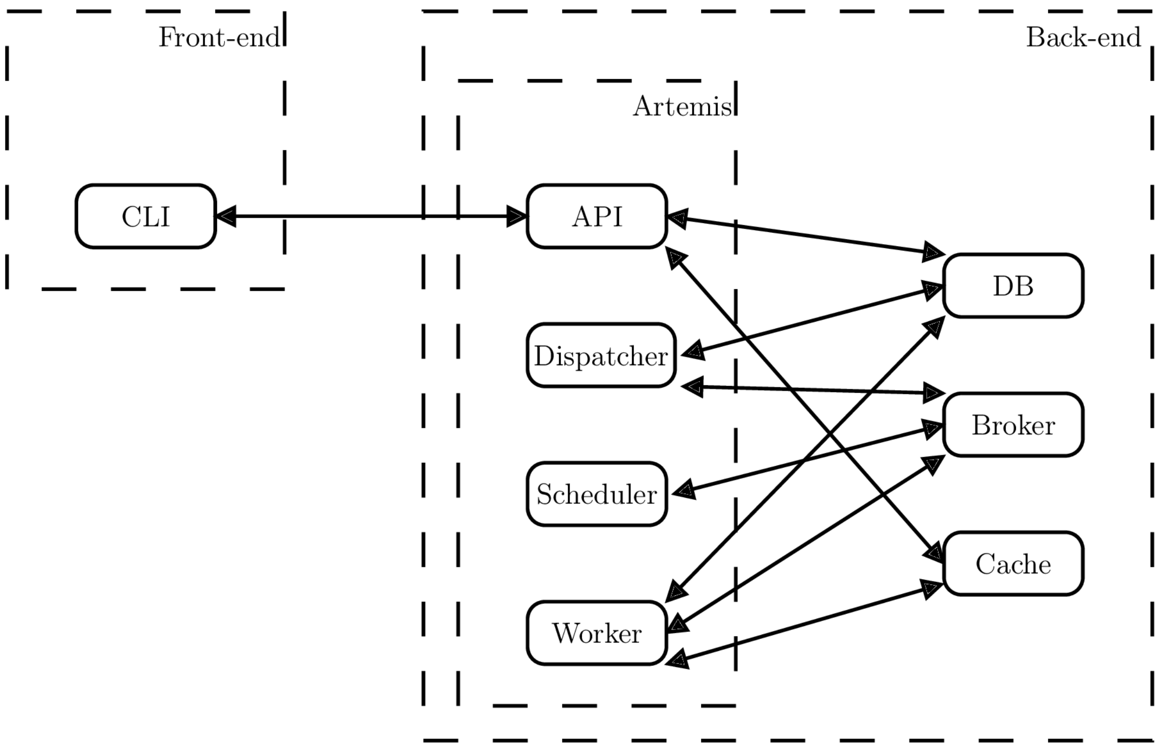

Figure 3.2: Diagram of Artemis architecture

3.2.2 Data Model

The data storage solution consists of a relational database — a type of database that, as described by Sai Sumathi [15], uses a collection of tables to represent both data and the relationships among those data allowing the identification and access to related data. The database stores the current state of the service.

3.3 Guest Request

A guest request represents a request made by a client to provision a guest machine, as well as the corresponding state of the machine.

The table `guest_requests` contains data identifying the specific guest request (`guestname`), requested environment (machine's features, configuration serialized in `_environment` field), the user owning the guest (`ownername`), selected pool (`poolname`), SSH connection information (`ssh_keyname, ssh_port, ssh_username`), and if successfully provisioned, the machine's IP address (`address`).

3.3.1 Guest Request Life-Cycle

The provisioning process is implemented as a series of discrete non-blocking tasks. Each task handles a step of the provisioning, updates the guest state and dispatches the next task in the series.

This process is highly configurable in order to facilitate flexible testing in numerous environments.

Routing

Upon creation, the guest request enters a state called `routing`, and the `route_guest_request` task is added to the queue. Routing is the first step of the process of acquiring a guest machine. This is the most customizable and scriptable part of the process, where, according to defined rules, pools, which satisfy the request, are selected. Furthermore, at this stage preference rules are evaluated, selecting the 'most suitable pool' to be used in the provisioning process. If the routing fails, the guest request enters the `error` state, and provisioning concludes at this point.

Acquiring guest

After routing, the guest is switched to `provisioning` state, and handed over to the `acquire_guest_request` task calling the driver of the chosen pool to attempt to obtain a machine as selected in the routing step. If the driver returns a machine instantaneously, the process proceeds onto the preparation stage. In practice, however, this process is not nearly instantaneous, and takes a significant portion of time to accommodate this wait until the guest becomes ready, the guest request is switched to the `promised` state (as it was promised by the driver to be ready eventually), and a task to update the state is scheduled. In case of a failure, the guest request falls back to the routing stage and is retried.

The `update_guest_request` task queries the driver on the current state of the guest. If the guest was successfully acquired, the preparation task is queued, otherwise this task is re-scheduled, and re-checks the state again later. In case of a failure, similarly to the `acquire_guest_request` task, the progress is discarded, and provisioning is restarted from the routing stage.

Preparation

During the preparation stage, by default the `prepare_verify_ssh` task verifies the availability, and proper configuration of SSH connection to the guest. This task can by configured to be skipped.

If a post-installation script is set, and the driver does not support running post-installation script natively, the `prepare_post_install_script` task is queued, which connects to the guest over SSH, and executes the script.

The last task in this stage is `guest_request_prepare_finalize_post_connect`, which marks the guest as `ready`, updates provisioning metrics, and schedules a `guest_request_watchdog` task, the purpose of which, unlike its name might suggest, is not to verify the availability, and proper operation of the guest, but rather it serves a specific function, where certain providers (drivers) require the guest reservation to be periodically extended to prevent premature termination.

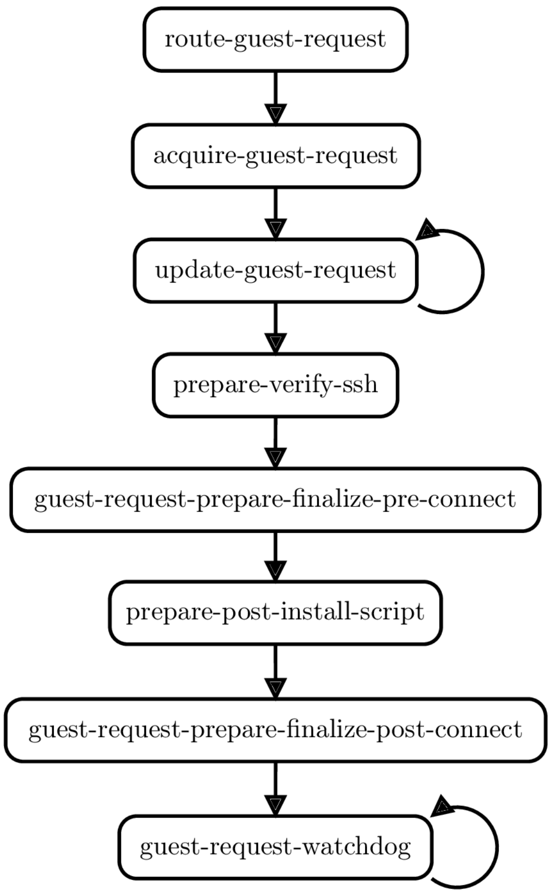
route-guest-request
acquire-guest-request
update-guest-request
prepare-verify-ssh
guest-request-prepare-finalize-pre-connect
prepare-post-install-script
guest-request-prepare-finalize-post-connect
guest-request-watchdog

Figure 3.3: Chain of provisioning tasks

Guest Release

Compared to the guest provisioning, the guest release is a notably simpler process. Upon the receipt of the guest delete command, the guest's state is set to `condemned`, and a `release_guest_request` task is queued.

During the execution of the task, the driver is instructed to release the guest and deletes the guest request entry in the database.

3.4 Tasks

Tasks define units of operations carried by the Artemis workers. A message broker is used to relay and queue the messages identifying tasks which are needed to be performed and dispatch them to a suitable worker.

Upon receipt of the message, the worker performs the operation defined by the task and logs the result. Optionally, it can dispatch a new task to the broker and then awaits further tasks from the broker.

Due to the synchronization requirements, some tasks cannot be dispatched immediately but only after a successful commit to the database because the next task might require the guest to already be in a certain state. In such a case, a **task request** can be created as a part of the database transaction modifying the guest's state. The task request identifies the task that would need to be dispatched to the broker, as well as any accompanying arguments passed along to this task.

New task requests are periodically queried from the database by the **dispatcher**, which submits these tasks to the broker and removes the entries from the database.

3.5 Components

As mentioned in the architectural overview (3.2), Artemis server consists of multiple inter-operating components. This section will provide deeper look into the inner workings and structure of the relevant components.

The server source code is contained in `server/src/tft/artemis/` directory in the repository.

3.5.1 API

The API module source is located in the `api` directory. The API is built with the use of Molten Framework[5]. The framework provides abstraction on top of Python Web Server Gateway Interface[6] for building HTTP APIs.

The framework [3] introduces multiple concepts to simplify the writing of the API:

- **schema** — a schema of the request content;
- **handler** — a method handling specific route;
- **component** — manages objects that may be requested by a handler;
- **route** — endpoint, path, where a request can be made;
- **middleware** — a glue code for pre-processing requests;

As described by its documentation, key features the framework provides include **request validation**, which ensure only valid data are received by the handler. In addition, the framework performs **dependency injection**, which helps decoupling handler and its dependencies by making components easily swappable and individually testable.

The Artemis API implements managers for the different entities handled by it. This enables for better and more reliable backwards compatibility as the API implements all the different older/legacy versions of the API, so that even clients using older API revisions can leverage latest Artemis features and optimizations.

The typical workflow of API includes querying the database for the requested data in case of GET requests, or creating or modifying data, followed by a dispatch of a task to a broker or adding an entry to the transactional outbox (described in 3.5.2) for further processing.

3.5.2 Dispatcher

As Artemis often requires synchronization between database operations and the dispatch of tasks, it leverages transactional outbox[7]. The dispatcher periodically queries the database

[5]Molten Framework: https://moltenframework.com/

[6]Python WSGI: https://peps.python.org/pep-3333/

[7]Transactional outbox pattern: https://microservices.io/patterns/data/transactional-outbox.html

for new task and snapshot requests. If a new task request was committed to the database, the relevant task is resolved and a task is dispatched to the broker for a worker to pick up.

3.5.3 Worker

The worker is implemented as a simple wrapper script dispatching dramatiq worker, which then connects to the broker and subscribes to all or just specified queues and awaits new tasks. Upon receipt of a message, the worker resolves and executes the task contained in the message. The tasks are closer described in 3.4.

3.5.4 Database Initialization Script

Implemented in `scripts/init_db_content.py`,

3.6 Metrics

Metrics are quantitative measures that provide insights into various aspects of software operation, performance and may provide insights helping identify potential areas for improvement. They play a crucial role in early anomaly detection and warning, and help ensure smooth and reliable operation of a service.

During normal operation, Artemis collects metrics primarily focused on the provisioning process, guests and pools in order to assist with catching, diagnosing and generating alerts for potentials issues. The metrics are exposed on a HTTP endpoint by the API in a format compatible with and scraped by Prometheus[8].

[8]Prometheus: `https://prometheus.io/`

Chapter 4

Design

In Chapter 2, we investigated various approaches employed to optimize the time lapse between requesting a virtual machine and receiving a provisioned machine for the end user. Moreover, Chapter 3 introduced the Artemis provisioning service, the context in which it is used, and the provisioning process as executed by the service. This section delves into an in-depth analysis of the current use cases for the provisioner and the design of enhancements to the existing provisioning process, aiming to augment the perceived responsiveness and efficiency of the service.

4.1 Requirements

Currently, Testing Farm executes each test plan in a separate fresh environment. This results in a new machine being provisioned to ensure that each test is run in isolation, preventing any potential interference from other tests or lingering artifacts from previous test runs. This approach, however, introduces a time delay associated with provisioning a new virtual machine, which is especially noticeable as the service tries to utilize the cheaper spot instances[1] many IaaS providers provide as a way to sell their spare compute capacity at a significantly discounted price.

The goal is to introduce a mechanism transparent to the user to reduce the time the test workload waits for a machine to become available. Furthermore, as a non-trivial number of test scenarios are non-destructive — container testing. In these cases, we would like to be able to mitigate the overhead of setting up a container runtime and re-use existing machines to run multiple of these tests as containers run in isolation and do not influence the environment of the executor machine and thus cannot influence other container tests that would run in the same VM.

From the techniques discussed in Section 2.2, the methods described by [10, 12] to reduce delay from loading VM image templates cannot be applied to our use-cases as we do not maintain the underlying infrastructure. Therefore, we have to rely on the providers to do their best to optimize this part of the provisioning process. Similarly, the approach used by [16] is not a feasible solution in our system as it is only suitable for systems and services where a fast and efficient replication of a specific application is required. In our case, the application being tested changes with every execution and as such replicating snapshot of initialized empty VMs could provide only negligible if any benefit.

[1] Amazon EC2 Spot Instances: https://aws.amazon.com/aws-cost-management/aws-cost-optimization/spot-instances/

The last technique discussed [9] proposes keeping initialize and running VMs in a cache and used these to satisfy requests. A similar approach can be replicated in the Artemis service as most tests are executed in a small number of compatible instance types. It would therefore be possible to provision a number of VM instances, keeping them in a cache and then be able to almost instantaneously serve new provisioning requests with one of these machines, if compatible with the requested machine.

Additionally, such an approach can be extended to serve as kind-of resource pool (as described in Section 2.3). After execution of non-destructive tests, the machine is left in a state suitable to run additional test plans. Currently, such a machine is completely released, however, it would be possible to return this machine to a pool of existing machines, which can be used to immediately serve new requests.

This hybrid resource pool-cache consisting of compatible machines was decided to be referred to as a 'shelf' in this work. The work will be further split into two parts building on top of each other:

1. Shelving: A pool of compatible, re-usable machines;
2. Pre-provisioning: Building the foundation for triggering the provisioning and shelving of a multiple machines.

As this was a long-awaited feature, the team already had an idea of what these features should entail[2].

4.1.1 Shelving

The primary outcome expected from the shelving feature is to extend the guest provisioning and release processes, as described in 3.3.1, with a transparent mechanism to serve new requests with already existing machines, as well as, returning machines to a shelf. Since there is no mechanism to determine whether the machine was modified or not, this decision will be left on the user to make.

Important thing we need to ensure is active monitoring the state of the shelved machines as sometimes machines can be taken away from us without notification, especially if we rely on cheaper-priced spot instance types[3]. For this reason, a periodically-running watchdog checking the health of the machine must be included in the solution.

Furthermore, there needs to be a mechanism to administer the shelves and shelved guests using the Artemis CLI. The required actions include:

- listing all shelves,
- getting information about a specific shelf,
- creating a shelf,
- removing a shelf,
- listing shelved guests,
- releasing a shelved guest,

[2]Shelving and pre-provisioning issues: https://gitlab.com/testing-farm/artemis/-/issues/118, https://gitlab.com/testing-farm/artemis/-/issues/117

[3]AWS EC2 Spot Instances: https://aws.amazon.com/aws-cost-management/aws-cost-optimization/spot-instances/

- configure maximum number of guests on a shelf.

It is also important we are able to monitor the behaviour of the shelves. For this reason a number of metrics are required:

- Shelf usage — the number of guests on a shelf;
- Shelf hits — tracking the number of guests served from a shelf;
- Shelf misses — the number of guests failing-over to normal provisioning;
- Forced removals — the number of guests forcefully removed by a user;
- Dead guests — the number of guests in a shelf that stopped responding (died);
- Removals — the aggregate number of guests removed from a shelf for any reason.

4.1.2 Pre-provisioning

Pre-provisioning is building on top of shelving as described in 4.1.1. The expected outcome is a mechanism, which would be utilized to trigger the replenishment of a shelf and serve as an extensible foundation for further automation of this process if desired later on.

This feature would consist of a mechanism to create a specified number of guest machines according to a specified guest template and subsequently release them to a shelf on provisioning completion. Initially, only manual trigger is to be implemented. This would mean extending the shelf operations with an additional one allowing the user to specify the parameters required for pre-provisioning.

4.2 Shelving

This section deals with the design of the shelving mechanism and its principle of operation. As outlined in the requirements, described in Section 4.1.1, this involves modification of the current provisioning pipelines. At this point, no partitioning of these shelves will be implemented, which results in certain restrictions for what guests would be considered suitable for shelving.

4.2.1 Modifications of the Guest Request Life-cycle

In order to facilitate the required functionality, modifications to the guest request's life-cycle are required. New operations have to be performed before the original guest provisioning and release flows. Using a shelf allows for these original processes to be bypassed. The idea behind these changes is illustrated by the Figure 4.1.

This approach results in the creation of two new tasks, which would be executed during the standard provisioning and guest release processes. Furthermore, to distinguish an active guest from a shelved guest a new state for these guests has to be added.

4.2.2 Provisioning

First, in order to bypass the provisioning and use a shelved guest, it is required to attempt to pick a compatible guest from a shelf before performing routing to an available pool for

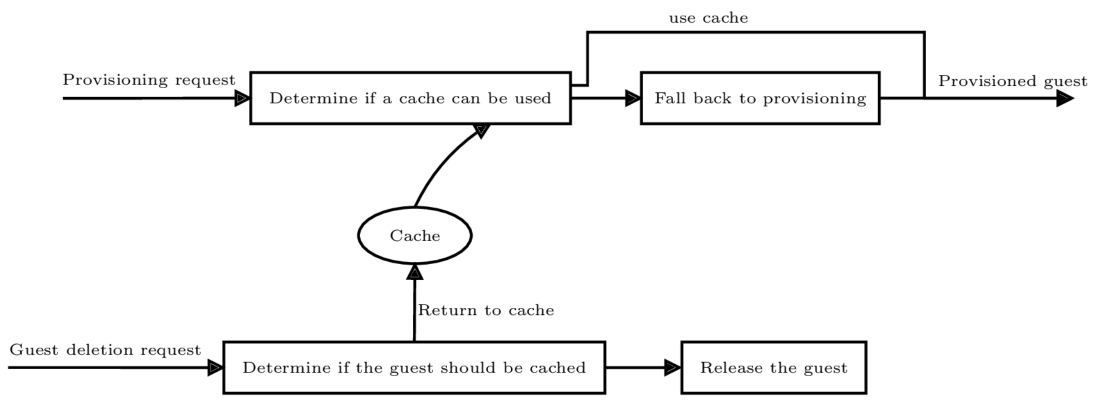

Figure 4.1: Expected changes to the provisioning flow

provisioning. Since this mechanism is supposed to bypass the provisioning entirely, a new task is inserted at the start of the provisioning pipeline.

The shelved guest selection has to be a robust mechanism returning a guest only if certain criteria are met:

- The guest is present in the specified shelf;
- The guest is compatible with the request.

4.2.3 Release

In order to be able to use guests stored in a shelf, there needs to be a mechanism to fill up the shelf. Therefore, if a shelve-able guest is being removed, instead of completely releasing it, it can be shelved and later re-used.

However, there are several restrictions placed on this mechanism. First of all, it is necessary to control the maximum number of guests in order to prevent runaway resource and money usage if too many guests were to be provisioned and stored on a shelf. Thus a verification of the number of guests already on a shelf does not exceed the configured maximum of allowed guests has to be performed.

Additionally, since no partitioning of shelves is being implemented at the current time and vast majority test scenarios utilize standard pre-configured guest types, it is necessary to ensure no special guests are being shelves as these would be of very little if any use. To avoid wasting resources associated with shelving such guests, the shelving task has to perform validation no additional constraints for the environment were set or a special post-installation script to configure the machine was provided by the user.

Last condition is that the guest must be operational at the time of shelving. The user can cancel a provisioning request before the guest is fully provisioned. Executing a guest cancellation command immediately marks the guest for removal and any ongoing provisioning is immediately stopped. Therefore such an incomplete guest is not suitable for serving requests.

If any of these conditions were to be violated, a regular guest release should be scheduled instead.

4.2.4 Watchdog and Monitoring

Since the guests in shelves can be long(er)-lived, it is necessary to monitor their state, and ensure that it was not destroyed by an external action and can still be accessed. For this purpose, a periodic task is to be scheduled for the duration of the guest's presence in the shelf, ensuring the guest is still in a operational state, and disposing of any dysfunctional guests.

4.2.5 Shelf Representation

We need to be able to represent a shelf in the system. Since there are plans for individual users to be able to bring their own infrastructure, it needs to be possible to isolate resources between users. Consequently it is not possible to share shelves between different users of the service, and therefore only one user can own and access any given shelf as well as any related resources. The shelf itself needs to be capable of holding guests, which, since we will be selecting them from and returning to a specific shelf, the guest will be tied to a specific shelf for the entire duration of its life-cycle.

Shelf therefore represents a new entity being introduced to the system. It is owned by a user, however a user can have a number of separate shelves for different purposes. The shelf is also capable of 'owning' guest requests — or rather a guest can belong to a shelf. This relationship is illustrated by Figure 4.2.

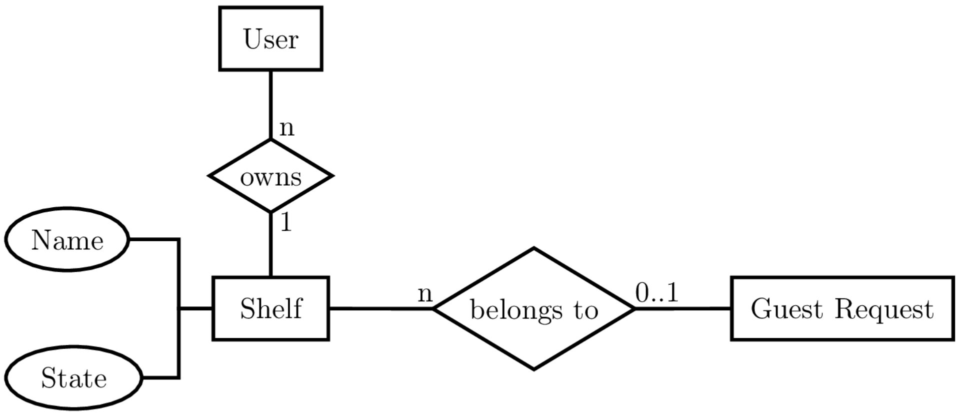

Figure 4.2: Entity-relation diagram of a shelf

4.2.6 Shelf Configuration

As specified in the requirements, there needs to be a possibility to manage and configure the shelves and its properties.

API

The user-facing management interface is implemented as a REST API by Artemis. It will, therefore, be necessary to extend this interface to allow the user to submit operations related to the shelf management.

The shelving-specific methods will be implemented under the `/shelves/` endpoint.

First and foremost, it is necessary for the user to list the resources under their management. For this reason, submitting a `GET` request to the `/shelves/` endpoint should return a list of shelves managed by them. Subsequently, provided the name of a specific shelf, the user might desire to obtain only parameters of a specific shelf. This purpose will be served by the `/shelves/<shelfname>` endpoint. Both of these endpoints would have to fetch the data of all or just the specified shelf from the database and relay them as a response to the client.

Another important operation is the shelf creation. This operation should be implemented as an endpoint responding to `POST` requests to the `/shelves/<shelfname>` endpoint. As there are no complex operations associated with the creation of a shelf, upon receipt, a new entry in the shelves table can be created for a shelf with the specified name. The shelf can be immediately active and used.

Shelf deletion is, however, a more complex problem as a shelf might have guests associated with it. For this reason, the API component itself should not handle this process and it is delegated to the workers to perform the clean-up and deletion. The API changes the state of the shelf to `condemned` to indicate it is no longer supposed to be used and creates a task request to dispatch the shelf removal task.

Last but not least, there needs to be a way for user to instruct the service to release a shelved guest. In order to preserve original functionality of the `/guests/` endpoints and isolate shelves from the rest of the guest management, the `/guests/<guestname>` endpoint should not accept `DELETE` requests for shelved guests. This helps prevent unintended removals and releases of shelved guests. For this reason, a dedicated endpoint is required to schedule the release of shelved guests. A `DELETE` request to the `/shelves/guests/<guestname>` endpoint will schedule the release of such a guest.

The remaining operations, such as adjusting the shelving-related variables will be serviced by existing endpoints. The newly-created endpoints are summarized in the Table 4.1.

Endpoint	Method	Description
`/shelves/`	`GET`	List available shelves
`/shelves/{shelfname}`	`GET`	Get shelf info and properties
`/shelves/{shelfname}`	`POST`	Create a shelf
`/shelves/{shelfname}`	`DELETE`	Remove the specified shelf
`/shelves/guests/{guestname}`	`DELETE`	Delete guest from shelf

Table 4.1: Shelf management Endpoints

4.3 Pre-provisioning

As specified by the requirements (Section 4.1.2), at the moment there was no desire in implementing a full-blown inventory management system. Instead, the plan is to introduce a mechanism to allow the user to fill a shelf with a user-specified guest types. Multiple approaches were evaluated for this purpose.

First possible way to approach the mechanism is to entirely decouple this functionality from the server in the form of an external application interfacing with the API. This was envisioned as a custom subcommand implement in the CLI. The client would begin provisioning a number of guests as specified by the user, wait until the provisioning completes and then automatically release the guests leveraging mechanisms introduced by shelving. Although sufficient for the manual guest pre-provisioning, this approach limits the options for future extendability and automation. Furthermore, such an approach might introduce possible unwanted reliability issues as it would be difficult to resume the process if the client was to disconnect or outright crash.

Another approach is implementing a separate server component running alongside workers. This would help mitigate the issues mentioned with the CLI-based approach while still maintaining isolation from the rest of the services, and thanks to the deeper integration with the rest of the system would enable possible robust automation in the future. However, it would provide no benefit compared to the last approach while consuming additional resources as it would have to be deployed as a separate component of the service.

The last explored approach is based on tasks. The idea is to create a separate task, which would be dispatched upon receiving a pre-provisioning request. The task would be responsible for the creation and dispatching provisioning for new guests. As a task, it can be easily dispatched by the API and other components of the service through already established mechanisms and allow for a relatively simple extension of functionality later on, if desired.

After a discussion with the project owner, the task-based approach was deemed best suited for our use-cases and therefore, the design of the final solution is based on this approach.

4.3.1 Pre-provisioning Process

The pre-provisioning process is going to be implemented as a new task. This task can be be triggered from multiple sources, such as a shelf reaching a minimum guest count threshold or a manual user pre-provisioning request. From these only the manual trigger will be implemented as a part of this work. The schema between the interaction between the triggers, the pre-provisioner and shelves are illustrated by the Figure 4.3.

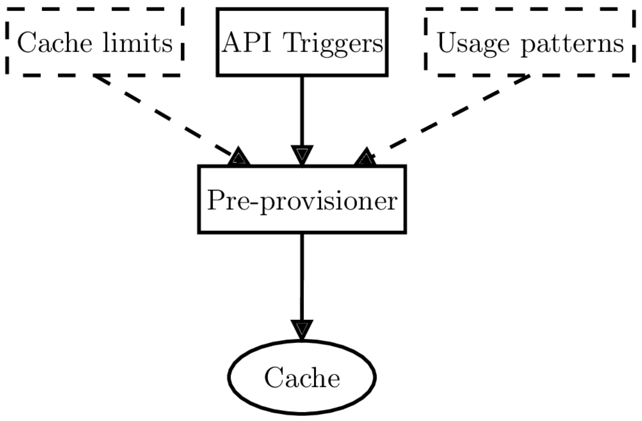

Figure 4.3: Pre-provisioner as a task with multiple triggers

The task itself will have to accept a template of a desired guest type, the target shelf and the number of guests requested. The task will then create entries for these new guest requests and dispatch tasks to begin the provisioning for these guests.

Given the first step of the provisioning pipeline is an attempt to lookup a compatible guest in a shelf, the shelving mechanism will have to be mitigated to not provide an existing guest.

Additionally, a mechanism for dispatching the shelving of the newly-provisioned guests is required. An option would be to have a task periodically refreshing and waiting for the current guests to complete provisioning and then move them to the target shelf. However, this introduces addition overhead associated with running additional tasks on workers. A better solution appears to extend the provisioning mechanisms with the ability to dispatch a task upon provisioning completion automatically. Then the new guest requests could be configured to be automatically released and the shelving would be handled by the regular shelving mechanisms described in Section 4.2.3.

4.3.2 User Interaction

The only mechanism to trigger the pre-provisioning implemented as a part of this thesis will be the manual triggering. This will consist of an API endpoint `/shelves/<shelfname>/preprovision`, which accepts POST requests containing the necessary information to prepare guests.

Chapter 5

Implementation

This chapter provides an overview of the inner workings of Artemis and the specifics of the implementation of the solution as designed in the Chapter 4. The initial sections of this chapter focus on the description of the current implementation of certain mechanisms and abstractions in Artemis in order to be able to implement the shelving and pre-provisioning mechanisms and then, building on this knowledge, describes the implementation of these mechanisms as well as the modifications that were necessary to facilitate these processes. The implementation was done in two phases. The first phase consisted of the shelving mechanism to facilitate the re-use of machines. The second phase built on top of the first one, implementing a basis for the pre-provisioning mechanism.

5.1 Artemis Project Structure

Artemis is implemented in the Python programming language[1] and utilizes Poetry[2] as its dependency manager. The main repository contains two modules: the server and co-developed CLI used to manage and interact with the server.

The repository layout is described in the Table 5.1.

Directory	**Description**
`server/`	Root of the server module
`server/configuration/`	Default location of the server configuration
`server/src/tft/artemis/`	Source code of the server module
`cli/`	Root of the command line interface module
`cli/src/tft/artemis_cli/`	Source code of the CLI module

Table 5.1: Repository layout

5.2 Working with the Database

Artemis utilizes SQL toolkit and object-relational mapper SQLAlchemy[3]. This python library provides abstraction over the underlying database management system and abstracts away raw SQL statements and enables the developer to manipulate the data as if they

[1]Python: https://www.python.org/
[2]Poetry: https://python-poetry.org/
[3]SQLAlchemy: https://www.sqlalchemy.org

were native Python objects. In the production environment, the database is backed the PostgreSQL database management system[4], however thanks to the ORM framework, it is possible to utilize SQLite[5] during the first stage of unit testing as it does not require a separate server.

The database ORM definition and wrappers for safe data manipulation are implemented in `server/src/tft/artemis/db.py` using the SQLAlchemy framework along with Alembic to handle automating of creation and executing database migrations.

Table Definition

SQLAlchemy's [5] declarative mapping is used to construct mappings for database objects. This is done by constructing a base class called `Base` using the `declarative_base()` function from which then the mapped classes are inherited from.

The classes are then mapped on a table specified by the `__tablename__` with columns specified as `Column` objects assigned to attributes of this class and mapped to columns of the same name. SQLAlchemy allows to specify the column's data type, which maps Python type to the most suitable column type available on the target database. Some notable types include:

- `Enum` — a type for mapping enumerations;
- `Integer` — maps Python's `int` type;
- `String` — maps string and character types.

Furthermore, it is possible to specify the column constraints. Most notable constraints can be specified directly as keyword arguments of the `Column`:

- `nullable` — specifies if the column's value can take up on an empty (`NULL`) value;
- `primary_key` — whether the column is the table's primary key;
- `unique` — whether each value in the column must be unique.

It is also possible to specify additional constraints as `Constraint` objects passed to the `Column` constructor. From these, important is the `ForeignKey` defining a dependency between two columns.

The `Column` usage is `Column(type, *constraints, primary_key=<bool>, nullable=<bool>, unique=<bool>)`.

As a simple example, a mapped table named 'orders' in the database, consisting of an integer primary key `id` uniquely identifying the specific order, integer foreign key user identifying the user account creating the order and a string `state` storing the state of the order, which can take on one of 'pending', 'shipped' or 'completed' values. This table can then be defined as follows:

```
@dataclasses.dataclass
class OrderState:
    PENDING = "pending"
    SHIPPED = "shipped"
```

[4]PostgreSQL: https://www.postgresql.org/
[5]SQLite: https://www.sqlite.org/index.html

```
    COMPLETED = "completed

class Order(Base):
    __tablename__ = "orders"

    id = Column(Integer(), primary_key=True, nullable=False)
    userid = Column(Integer(), ForeignKey('users.id'), nullable=False)
    state = Column(
        Enum(OrderState),
        nullable=False,
        server_default=OrderState.PENDING.value
    )

    user = relationship(User, back_populates='orders')
```

Executing Database Statements

Building on top of the Artemis's error handling, wrappers for safely manipulating database are used.

For querying the database a class `SafeQuery` is used, allowing to build up `SELECT` queries using the SQLAlchemy's abstraction. The class is typically instantiated using the `SafeQuery.from_session(session, klass)` method, allowing to specify the database session and the queried table.

To add a `WHERE` clause to the statement, the `filter` method is called with the desired criterion as an argument. The mapping between SQL condition operators and ORM query constructs is described in table 5.2.

`column=value`	`MappedColum == value`
`column>value`	`MappedColum > value`
`column<value`	`MappedColum < value`
`column>=value`	`MappedColum >= value`
`column<=value`	`MappedColum <= value`
`column<>value`	`MappedColum != value`
`column BETWEEN lower AND higher`	`MappedColum.between(lower, higher)`
`column LIKE value`	`MappedColum.like(value)`
`column IN (values, ...)`	`MappedColum.in_([values, ...])`

Table 5.2: SQL conditional operators and their ORM equivalent

To obtain the result of the query, a call is made to one of the available methods according to the expected result:

- `SafeQuery.one()` — returns exactly one row or error;
- `SafeQuery.one_or_none()` — returns one row or `None` if there is no match, error otherwise;
- `SafeQuery.all()` — returns a list of all matching rows.

An example of usage:

```
result = SafeQuery.from_session(session, Order) \
    .filter(Order.state == OrderState.PENDING) \
    .all()
```

The statements modifying contents of the database are built from `insert`/`upsert`, `update` and `delete` methods. For statements changing the row's values, the values are specified using the `values()` method. Except for the insert/upsert statements, selecting specific rows to modify can be achieved by using the `where()` method with condition.

Examples of such statements:

```
# INSERT statement
insert(Order).values(id=4031, userid=2)
# UPDATE statement
update(Order).where(Order.id == 4031).values(userid=1)
# DELETE statement
delete(Order).where(Order.id == 4031)
```

Constructed statements are executed using one of `execute_db_statement` or `safe_db_change` wrappers. These handle the execution of, logging of the operation and any exceptions possibly raised by the operation. The difference in usage is that the `safe_db_change` function is used to perform changes, which must be synchronized with other actions by issuing an explicit `COMMIT` after executing the statement.

Database Migrations

Artemis's database migrations are implemented using SQLAlchemy's Alembic tool [1]. The tool is used to apply upgrades or downgrades of the database schema in order to be able to move between Artemis versions. Additionally, alembic is able to automatically generate migration scripts based on the difference between the latest schema revision present in the database and the metadata built from the database as defined in the code.

To execute migrations to the latest revision, the following command can be executed:

```
alembic upgrade head
```

Afterwards, a new revision can be generated using:

```
alembic revision --autogenerate -m "New migration"
```

The generated script, however, may not capture all the changes made and may therefore often require additional manual modifications.

The script consists of variables `revision` identifying the version this script revises the database to, `down_revision` identifying the version this script revises from. The migrations themselves are implemented by the function `upgrade` and `downgrade` for up and down revision, respectively.

5.3 Task structure

Each task is implemented in its own source file present in the `server/tft/artemis/tasks` directory and follows a common structure consisting of 4 parts.

First a task-specific workspace class is defined. This class is inherits from a general `Workspace` class defined in the `tasks` module initialization script (`tasks/__init__.py`) should store necessary task-specific variables and define the steps of the task as methods. The steps of the task are implemented as methods with no return value within the task-specific workspace class. Each step should be decorated with the `@step` decorator.

A simple example of the structure:

```
from . import Workspace as _Workspace

class Workspace(_Workspace):
    """
    The task-specific workspace
    """

    TASKNAME = 'sample-task'
    # Task-specific variables
    # ...

    @step
    def entry(self) -> None:
        """
        Begin the process by nice logging and loading required data
        """

        self.handle_success('entered-task')

        # Load data
        # ...

    # Additional steps
    # ...

    ...
```

A class method within the workspace is used to create and initialize the workspace and execute the chained steps of the task. This method should return the final task result.

This can be implemented similarly to:

```
class Workspace(_Workspace):
    ...

    @classmethod
    def sample_task(cls, ...) -> DoerReturnType:
        # Create and initialize the workspace
        workspace = cls.create(...)

        # Execute steps
        final_result = workspace \
            .entry() \
            # Additional steps are chained as calls to the corresponding
            # methods
            .final_result

        return final_result
```

The final part of the task is the entry point function that wraps the task execution in the `task_core` function, which handles common task-related functionality like logging and session management. The entry point is always decorated with the `@task` decorator.

Example:

```
@task
def sample_task(*args) -> None:
    task_core(
        cast(DoerType, Workspace.test_task),
        logger=get_*_logger(Workspace.TASKNAME, _ROOT_LOGGE, ...),
        doer_args=args,
        session_isolation=True
    )
```

This structuring increases the modularity of the code, as well as the readability. Another major consideration is the testability. Structuring tasks as shown enables writing fine-grained unit tests testing the behaviour of individual steps.

5.4 Run-time Variables

Artemis uses user-configurable run-time variables implemented under the name of *Knobs*. These allow the user to configure certain properties of Artemis's components and provisioning processes, such as the number of worker threads, enabled pools, etc.

Knobs can utilize various sources for their values, organized in a hierarchy:

1. A database-backed storage implemented by the `KnobSourceDB`
2. A value read from environment variable implemented by the `KnobSourceEnv`;
3. A constant or default value represented by `KnobSourceDefault` or `KnobSourceActual`;

In addition to knobs setting values globally for all components, under certain circumstances it is desirable to be able to configure the behaviour of different pools utilizing the same driver. For this *per-pool* knobs exist, allowing to use knob names parametrized with the pool's name, modifying the behaviour of a specific resource pool only. The pool-specific values can only be loaded from environment variables and database, and if the value cannot be determined, the value for driver (instead of a specific pool) is used.

Usage

An example of knob usage:

```
KNOB_EXAMPLE = Knob(
    'example',
    'Helpful description of the knob.',
    has_db=True,
    per_pool=True,
    envvar='ARTEMIS_EXAMPLE',
    cast_from_str=int,
    default=0
)
```

Since the specified knob is a 'per-pool' knob, a `Pool` object or the name of the pool needs to be passed to the `get_value` method. The value of the knob can be obtained by:

```
    KNOB_EXAMPLE.get_value(pool=pool)
```

5.5 Error Handling

Due to Artemis's architecture, there is a need to be able to safely propagate and log errors throughout the code as an unexpected or improperly handled exceptions could potentially leave the system in an unexpected state. For this reason, wrappers storing results and error or exception data exist.

Notably, to propagate the success or failure of an operation `Result` objects are used. This class, however, is not instantiated directly but the class-provided methods `Ok` and `Error` are used, which set-up the instance with additional information for faster resolution of whether the operation succeeded or not. These are provided by gluetool[6] library, which contains the code shared across Testing Farm projects.

To instantiate the `Result` object in case of a success, a call `Ok(success_value)` is used and similarly the object can be created in case of an error using the `Error(error_value)` call. Upon receiving the `Result` object, the caller/receiver can easily query the result of the operation by checking the `Result.is_error` properly, which relays the information whether the object holds a valid result value of a successful operation or information about an error. The value itself can then be accessed by calling the `Result.unwrap()` or `Result.unwrap_error()` in case of an error.

In addition to these, Artemis implements `Failure` class, which stores the information about a erroneous situation, including the information provided by a raised exception or the source of the failure (as another instance of `Failure`), as well as other information clarifying the context and source of the failure. These are used for logging purposes and the class itself also implements the logic to submit errors to the Sentry monitoring service[7].

5.6 Shelving

This section focuses on the implementation of the solution design as described in the Section 4.2.

5.6.1 Preparatory work

Multiple places were identified before or during the feature's implementation, which required modifications in order to be able to advance the work on the shelving mechanism.

[6]Gluetool: https://gluetool.readthedocs.io

[7]Sentry: https://sentry.io

Generalize 'per-pool' knobs

As it is necessary to be able to configure the behaviour of the shelves, the Artemis's knobs were the obvious mechanism to be used to facilitate this need (rephrase). The knobs, however, bore a limitation with respect to their usage, which was limited to either global variables, or scoped variables corresponding to pools (per-pool). This 'per-pool' behaviour thus had to be generalized for any kind of resource.

The way the per-pool knobs are used is that the knob's consumer can specify one of `poolname` or `pool` parameters, which either are directly a string name or the string can be obtained from the passed object. This is then used to parametrize string identifying environment variable in the format '`<envvar_name>_<poolname>`' or database primary key identifying the knob in the format '`<knob_name>:poolname>`'.

After understanding the structure of implementation of knobs, the required changes become fairly simple. First, the option to pass pool object when obtaining the knob's value was removed (`pool` parameter of `get_value` function), leaving only the option to use `poolname` parameter to specify the name of the pool. This change was sufficient to effectively decouple knobs from pools.

To reflect this change in semantics, the per-pool knobs were renamed. As these knobs were now no longer tied to any specific entity, it was decided that the appropriate label for these knobs would now be 'per-entity'. As a result, the `poolname` parameter was renamed to `entityname`, as well as all `*PerPool` back-ends were renamed to `PerEntity`.

Additionally, it was necessary to update all occurrences of now per-entity knobs to conform to the new usage.

Count Query

Additionally, the `SafeQuery` wrapper did not implement method to obtain the number of matching rows in the database, which is required in order to determine the number of currently shelved guests. Analogously to other operations, the `count()` method was implemented as a wrapper around the SQLAlchemy's native `count()` method by adding type annotations and decorating it with the `@chain_get` decorator:

```
class SafeQuery(Generic[T]):
    ...

    @chain_get
    def count(self) -> int:
        return cast(
            Callable[[], int],
            self.query.count
        )()
```

5.6.2 Shelf Table

In order to be able to start working with shelves, related constructs had to be implemented before proceeding with any further work on the logic handling shelves or shelf-related guest operations. The shelf entity, described in Section 4.2.5, translates into the relational model used by Artemis into the table `guest_shelves`. The table is illustrated by the Figure 5.1.

The concrete implementation was done using a mapped class `GuestShelf` as described in Section 5.2. The class itself defines the string attributes `shelfname` and `ownername`, which

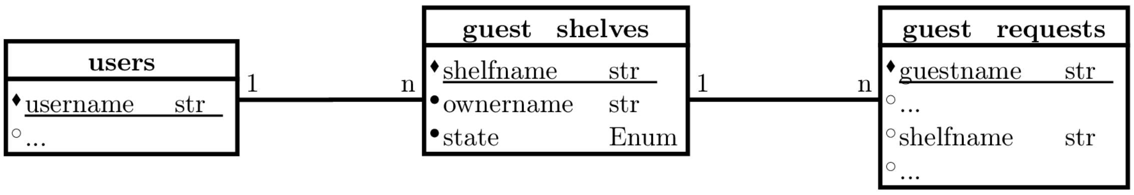

Figure 5.1: Shelf database diagram

identify the shelf itself (primary key) and the owner of the shelf. The `ownername`'s definition therefore contains a foreign key constraint, referring to the `users` table. Last attribute of the shelf is its state. This was chosen to re-use the `GuestState` enumeration as it already contains all the required state for the shelf.

Additionally, following the conventions set in the implementation, `create_query` helper method was added. This method serves as a wrapper for creation of `INSERT` queries for new shelves.

A new database migration script, bumping the revision to `3af7c26ec4f3` was generated in order to facilitate the creation of the table in the actual database.

5.6.3 Management

As outlined the requirements and design of the feature in the Section 4.2.6, being able to manage the shelves by the user is a crucial aspect requiring a working and stable API.

API

The API module, located within the `api/` subdirectory of the server's source, implements the REST API interface served as a part of the service. In order to manage the shelves a manager class for this new entity is implemented. A number of methods were required to perform the necessary operations with shelves as outlined in the requirements, as well as a schema class defining the response for data on shelves.

In order to be able to leverage the schemas provided by the Molten framework, a `GuestShelfResponse` class is defined using the `@molten.schema` decorator. The class is implemented as a Python dataclass[8] so that it can be instantiated in a user-friendly manner by passing the attribute values to its constructor. An additional helper method `from_db()` is defined to help instantiate the class from a database object — `GuestShelf`.

To obtain the list of available shelves the `get_shelves()` method is used. It utilizes `SafeQuery` to query the database for all shelves belonging to a particular user and validates the result to ensure an error was not returned. The results are parsed into a list of `GuestShelfResponse` objects, which are returned by the function. Similar function serves the `get_shelf()` method, which, as its name may indicate, queries the database for a singular specific shelf. It's return value is a single `GuestShelfResponse` object.

Another important operation performed by the manager is the shelf creation. This is implemented by the `create_shelf()` method. This is performed by creating an insert query using the `artemis_db.GuestShelf.create_query(shelfname, ownername)` helper. After inserting a new entry for the shelf, if no errors were raised, the shelf details are again queried using the `get_shelf()` method and returned back to the caller.

[8]Pyhton dataclasses: https://docs.python.org/3.9/library/dataclasses.html

Last operation with the shelf entity is its removal, this is implemented by the `delete_by_shelfname()` method. For the purposes of error reporting to the user, the shelf is first attempted to be loaded. If found, as it is a more involved operation, it's state is updated to 'condemned' to indicate it is no longer available for use and a removal task is requested. The implementation of the task is dealt with in the Section 5.6.3.

With all the required operations implemented, handler methods are required to handle new API endpoints to be exposed. The handlers handle have to ensure the user is properly authorized, so that they can access the requested resources. These are implemented as static methods of the manager as they do not require being version for the time being.

An example of shelf create handler:

```
@staticmethod
def entry_create_shelf(
    manager: 'GuestShelfManager',
    shelfname: str,
    auth: AuthContext,
    logger: gluetool.log.ContextAdapter
) -> Tuple[str, GuestShelfResponse]:
    # TODO: drop is_authenticated when things become mandatory
    if auth.is_authentication_enabled and auth.is_authenticated:
        assert auth.username

        ownername = auth.username

    else:
        ownername = DEFAULT_GUEST_REQUEST_OWNER

    return HTTP_201, manager.create_shelf(shelfname, ownername, logger)
```

To expose the new endpoints, versioned 'route generator' methods are used by Artemis. The routes are added to the generator for the next (so far untagged) version:

```
@route_generator
def generate_routes_v0_0_56(...) -> List[Union[Route, Include]]:
    return [
        ...
        Include('/shelves', [
            ...
            create_route('/{shelfname}',
                    GuestShelfManager.entry_get_shelf, method='GET'),
            ...
    ]
```

For a full list of endpoints refer to the Table 4.1.

Shelf Removal Task

The shelf removal is the most involved operation in shelf management as there may be guests associated with the shelf, be it directly shelved guests or still active guests to be returned to the shelf. Due to this reason, the removal process should be handled by a separate task executed by a worker and the API does not itself handle the removal process, rather marks the shelf for removal and requests this task.

To be able to properly remove a shelf we need to be able to ensure no guest remains associated with the shelf, otherwise it would not be possible to remove the shelf as its removal would result in the violation of the referential integrity of the database since there would have to be guests in relation with a non-existent shelf. To maintain the integrity of the system and perform a proper cleanup, after the shelf is switched to the 'condemned' state, a separate task performs this process. The steps are:

1. Load Shelved Guests:
2. Schedule Release of Shelved Guest Requests: In this step, the task schedules the release of all shelved guests associated with the particular shelf.
3. Remove Shelf from Guest Requests: The shelf is removed from any remaining guest requests.
4. Delete Shelf: Upon reaching this step it is finally possible to safely remove the shelf itself. A DELETE query is executing removing the shelf from the database table.

5.6.4 Server Configuration File

In addition to the API, it may be desirable to bootstrap shelves from the server configuration file. The configuration file is a YAML[9] file capable of defining a number of entities within the system. It is allows to quickly bootstrap the Artemis server as it possible to define all the necessary constructs for operation in the `server.yml`. The file is parsed and the database is populated by the database initialization script `scripts/init_db_content.py`. For an example of the configuration file refer to Appendix B.1.

To add support for the creation of shelves using this script, the JSON schema defined in `schema/common.yml` needed to be extended. Subsequently, the `config_to_db()` method can be extended by a loop iterating over defined shelves and executes an UPSERT query to insert new or update existing shelves. A shelf declaration follows the layout:

```
shelves:
  - name: shelfname
    owner: shelf_owner
```

5.6.5 Choosing Guest From a Shelf

First of the guest life-cycle modifications made is the ability to select a valid guest from a shelf. This action was implemented in a new task `guest-shelf-lookup`. In order to differentiate the stage of the provisioning the guest is in, and to be able to properly report and monitor this state, a new guest state is introduced: 'shelf-lookup'. This also helps mitigate potential issues if a different task attempted to modify the state as each step in the provisioning process corresponds to a specific guest state.

The task consists of number of discrete steps:

1. Entry: In this step the task attempts to load a guest request passed for processing. The guest request must be in the 'shelf-lookup' state, otherwise the task concludes with an error at this stage.

[9] YAML Ain't Markup Language: https://yaml.org/

2. Shelf Query: At this step the task attempts to load all available shelved guests belonging to the shelf specified by the guest request.
3. Select Guest: The guests are selected from the shelf in a random order and validation occurs, ensuring the guest, if selected, is compatible with the requested guest. At this step an attempt is made to find the first compatible guest.
4. Use Guest: If a guest was selected in the previous step, the state of this guest is copied into the new request and the new guest is switched to the 'preparing' state and a task is dispatched to validate the guest is indeed accessible. If no compatible guest was found, this step performs no operation.
5. Remove Shelved Guest Request: This step is also dependent on the result of guest selection. If no shelved guest was selected this step is skipped. Otherwise, at this point the selected guest is removed from the database in order to prevent other guest requests from being served by the same guest, and the task concludes at this point.
6. Shelf Miss: If no guest was selected, at this step the 'routing' task is dispatched, falling back to the full provisioning of a new guest.
7. Exit: This step only handles metrics and logging the information about task completion.

This task uses a repeatable read isolation level for its database session in order to mitigate race condition where a different guest request could be assigned the same provisioned guest.

The default guest state upon its creation was changed 'shelf-lookup' state to reflect the change in the provisioning process. Furthermore, the guest request manager component in the API module was modified to dispatch this task after the receipt of a new guest provisioning request.

These changes enabled the use of a shelved guest to serve new incoming guest requests. For a diagram illustrating the change in the guest's life-cycle refer to the diagram in Appendix A.

5.6.6 Shelving a Guest

Upon the cancellation of a guest, it's state is changed to 'condemned', which discards any further information on the provisioning progress of the guest and cancels any ongoing provisioning operations as the tasks would no longer be able to find a guest request in the expected state.

The guest request manager in the API component therefore needs to be modified to determine whether the guest was in a state suitable for being shelved (it is possible to shelve only 'ready' guests). At this point therefore happens the decision on whether to attempt to shelve the guest by dispatching the new 'return-guest-to-shelf' task or fall back to the original full release of all guest resources.

If a guest is determined to be in a suitable state to attempt shelving, the decision whether to keep a guest in a shelf and performance of the actual shelving is delegated to the 'return-guest-to-shelf' task. This task consists of the following sequence of steps:

1. Entry: Loads the guest request in question.

2. Load Valid Shelf: Attempts to load a shelf associated with the guest request. The shelf has to be in 'ready' state in order to be used. Otherwise it cannot handle new guests.

3. Load Shelved Count: If a valid shelf was loaded in the previous step, the number of currently shelved guests is loaded from the database as it will be important to determine if there is a space for the new guest.

4. Load Shelf Maximum Guests: At this step, the task obtains a value from the `KNOB_SHELF_MAX_GUESTS` knob as it holds the maximum number of guests allowed to be held in a shelf.

5. Return Guest: At this step, the number of guests currently shelved and the number of allowed guests in a shelf are compared, ensuring the shelf does not exceed its configured capacity. Moreover, if the guest specified hardware requirements or a post-installation script, it is considered a special guest, that should not be shelved as it is not expected to be requested as commonly and therefore would waste resources. If the guest satisfies the given conditions, its state is switched to 'shelved', and a watchdog task is scheduled. The logs the successful result and concludes at this point.

6. Dispatch Release: If it was not possible to shelve the guest, a task to release its resources is scheduled at this step.

5.6.7 Guest Watchdog

While shelved, the guest can become inaccessible if it is taken away or released outside the Artemis service. Although it is not as critical issue because the validation of machine's accessibility is performed before returning a guest from a shelf, detecting guests in this erroneous state can free up space in the shelf for operational guests. The proper operation of a guest machine can be tested by the same mechanism guests are verified during provisioning. An attempt to create an SSH connection is made, which, if successful, indicates the machine is fully operational, and can be used to serve an incoming provisioning request.

A periodic task names 'shelved-guest-watchdog' is therefore scheduled for every shelved guest, ensuring the machine is accessible and operational. For proper operation across wide range of environments, the task requires configurable timeout for the SSH connection timeout and a period between task's successive runs. These are implemented as knobs:

- `KNOB_SHELVED_GUEST_WATCHDOG_DISPATCH_PERIOD`: This is a knob period of the watchdog task. It stores a static value configurable by an environment variable or defaults to 5 minutes and applies to all shelves and pools.

- `KNOB_SHELVED_GUEST_WATCHDOG_SSH_CONNECT_TIMEOUT`: This is a knob specifying the maximum allowed period while waiting for an SSH connection to be established before considering it failed. The default value is 15 seconds as that is the same default value used for verification during provisioning. This knob can be configured for a different value for each pool and is backed by the database, therefore can be changed at runtime.

The task then performs the following steps:

1. Entry: Loads specific shelved guest request and the pool that provisioned the guest.

2. End If SSH Disabled: This task respects the skip_prepare_verify_ssh guest attribute, therefore, if set, it indicates the guest is not or should not be accessed over SSH and therefore this check should terminate at this point.
3. Load SSH Timeout: If SSH verification is enabled, the task proceeds to load SSH connection timeout value for the pool that was used to provision the guest.
4. Run Watchdog: This step utilizes the ping_shell_remote method in order to initiate an SSH connection to the guest. It uses the SSH options configured for the specific pool as well as the timeout value loaded in the previous step.
5. Dispatch Release: If the SSH connection attempt fails, the guest state is switched to 'condemned', the guest release task is scheduled to cleanup any resource left by the machine and the task exits with an error.
6. Schedule Followup: After a successful connection to the guest, the task reschedules itself with a delay specified by the KNOB_SHELVED_GUEST_WATCHDOG_DISPATCH_PERIOD knob.

5.7 Pre-provisioning

This section describes the implementation of the pre-provisioning mechanism. The implementation follows the design laid out in the Section 4.3.

5.7.1 Pre-requisites

Bypassing Shelf

In order to be able to pre-provision a guest, an actual new guest needs to be provisioned. Since a shelf is paired with a guest for its entire life-cycle, there needs to be a way to instruct the 'guest-shelf-lookup' task to bypass the shelf and instead force the provisioning of a new guest.

This is achieved by adding an additional attribute bypass_shelf_lookup indicating the task should skip querying the specified shelf. Instead the list of available guests is replaced with an empty list, resulting in no suitable guests to select from and therefore falling back to the full provisioning.

Dispatch Tasks on Provisioning Completion

As a pre-requisite to be able to schedule the shelving of successfully provisioned guests in an automated fashion, a mechanism to dispatch the task upon provisioning completion is required. This is implemented such that it is not tied specifically to this use-case, however can be easily re-purposed to serve other goals, such as running hooks to notify user of provisioning completion, as well.

The GuestRequest table was extended by an _on_ready storing a JSON representation of a list of tuples of (actorname, arguments). This structure is accessed through an accessor on_ready, returning properly deserialized structure and fixes type annotations.

The 'prepare-finalize-post-connect' task was further modified. As this is the task where the guest is switched to the 'ready' state, upon executing the state change, task requests for all of the specified tasks in the on_ready field are created.

Shelving Guest in an Arbitrary State

At this point, the 'return-guest-to-shelf' task accepted only 'condemned' guests. However, we need to be able to move guests directly from the 'ready' state to 'shelved'.

This was a relatively simple change, which consisted of adding an additional parameter for the task specifying the guest's expected state. Therefore, when executing the first step, and the task requests a guest request, the expected current state is read from a variable stored by the workspace.

5.7.2 Task

The main part of the pre-provisioning mechanism consists of a task named 'preprovision'. The task requires the name of the shelf the guests would be pre-provisioned for, the number of guests to be pre-provisioned, and a guest template JSON-serialized into a string.

Upon initialization, the task performs the deserialization of the guest template into a `GuestRequest` object. The task is executed as a sequence of a number of steps:

1. Entry: This step attempts to load the shelf the guests are supposed to be provisioned for to ensure the shelf exists and is ready to accept these guests.

2. Parse Environment: This step is responsible for deserializing the guest environment specification into an `Environment` object.

3. Parse Log Types: At this step the defined log types required for the machines are parsed into tuples of `(string_log_name, GuestLogContentType)` as log types are required in this format by the `GuestRequest.create_query()` helper. Additionally, this way a validation is performed and in case of invalid data present, the pre-provisioning would fail.

4. Create Guests: This is the step at which the guests are created. For each requested guest an INSERT statement is created with a unique UUID as the guest name and in accordance with the provided template. The guest is created with the `bypass_shelf_lookup` attribute set to `True` as well as the 'return-to-shelf' task in the `on_ready` field. Upon the statement's execution, the 'guest-shelf-lookup' task is requested in order to begin provisioning.

Upon the completion of the provisioning of guests, the 'return-to-shelf' task is automatically executed and therefore, any suitable guests would be shelved.

5.7.3 API

In order to facilitate the manual triggering of pre-provisioning throughout the Artemis's REST API, a new action shelf-related action and an endpoint had to be implemented.

The pre-provisioning is a new action related to shelves. Therefore, a new `preprovision()` method was implemented as a part of the shelf manager. This method re-uses the code originally used for the validation of guest requests before their creation as it is necessary to validate the guest template provided by the user. The validation was factored out into the `_validate_guest_request()` method to avoid duplication of functionality within the codebase. After the validation, the guest request data are serialized into a dictionary, which is then transformed into a JSON string, which can be relayed to the 'preprovision' task over the message broker. Along with the guest template, the requested guest count and name of the shelf the pre-provisioning is being requested for are relayed to the task.

Similarly to shelf management endpoints, the pre-provisioning endpoint (`/shelves/<shelfname>/preprovision`) requires its own handler, however in this case, due to the changes being made to the guest request's schema, the endpoint's handler needs to be versioned. The first version of the handler, introduced in the API version 0.0.56, is implemented by the `preprovision_v0_0_56()` method following the naming conventions established for the guest creation endpoint. The handler performs authorization validation and then calls the shelf manager's `preprovision()` method, passing the parameters provided by the user along with the JSON schema of the guest environment for the version 0.0.56.

The parameters required to make a pre-provisioning request are defined a schema:

```
@molten.schema
class PreprovisioningRequest:
    count: int
    guest: GuestRequest
```

The endpoint itself is then exposed in the API by adding a route to the route generator under '/shelves':

```
create_route('/{shelfname}/preprovision', preprovision_v0_0_56,
             method='POST')
```

5.8 Testing and Evaluation

In order to ensure quality and proper operation of the introduced changes as well as the preservation of the original functionality, rigorous testing is required. This section describes the testing performed and attempts to measure and evaluate the potential benefits brought by the implemented optimisations.

5.8.1 Testing

The preservation of the original functionality is verified by the pre-existing test suites consisting of multiple parts:

- Static Analysis — The code uses type annotations, which can be used to validate the correct usage of variables, functions, etc. within the codebase;
- Database Migrations Tests — These are simple tests used to ensure it is possible to safely and reliably perform migrations between any two revisions of the database;
- Unit Tests — A method by which smaller units of code are verified to conform to the expected behaviour;
- Production DBMS Integration — The database migration tests as well as the unit test suites are executed against an environment mimicking the production environment setup.

During the implementation of the described modifications, unit tests were also written covering almost the entirety of the new code. For example, the behaviour of individual tasks created in order to amend the provisioning process is tested at the level of individual steps and the configuration of these tasks is also verified in the process.

Furthermore, strict code formatting standards are enforced and all of the mentioned checks are expected pass as a part of CI pipelines before any changes can be merged into the main repository.

5.8.2 Provisioning Speed

Synthetic scenarios were executed in order to measure the potential impact of these optimisations. These look at the provisioning times when using the Artemis provisioning service without the support for shelving, with support but not used, and with the shelving used. The scenarios were executed for the `localhost` driver as well as against the t2.small AWS instances. The results from the testing are summarized in the Table 5.3.

Scenario description	**Average time to provision** [s]
`localhost` driver on Artemis without shelving support	19.47
`localhost` driver without using shelving	29.79
`localhost` driver shelved guests	19.96
AWS guests without shelving	263.23
AWS guests from a pre-provisioned shelf	20.35

Table 5.3: Average of 10 runs for different provisioning scenarios

Initial rounds of testing were focused on the comparison of the original and updated provisioning pipeline. The tests were execute with a 'simulated' `localhost` driver. As can be seen from the results, there is an addition 10 second penalty incurred from using Artemis version with a support for shelving. This finding can be explained by the fact that an additional task was added to the provisioning pipeline. The tasks are dispatched using a transactional outbox and are read by the dispatcher every 10 seconds from where stems the additional slowdown. The use of shelved guests is comparable in duration with the original implementation. This is due to the fact a step in the pipeline is bypassed and therefore the same number of tasks are required to be dispatched by the dispatcher.

Although this would indicate a 50% slowdown in the provisioning, looking at the closer-to-real-world figures from actual AWS guests, considering a guest was fully provisioned on average after 263 seconds, the 10 seconds of additional delay are not creating a considerable relative slow-down. The performance gained from pre-provisioned guests, however, more than makes up for the shortcomings, cutting the provisioning time to a mere 20 seconds on average.

Chapter 6

Conclusion

The primary objective of this thesis was to design and implement a mechanism intended to optimize the time spent waiting for a machine to be provisioned by the Artemis machine provisioning service.

A technique for the re-use of suitable systems, as well as a method to prepare these machines in advance was devised and successfully implemented. The project consists of two parts — shelving, and pre-provisioning building on top of the shelving mechanism and extending its abilities.

The shelving introduced the concept of 'shelves' — pools of (virtual) machines — and resulted in the modification of the machine provisioning flow such that an already provisioned machine can be returned to a shelf instead of being completely released and then used to serve a new provisioning request received afterwards.

The pre-provisioning mechanism enables preparing guests and releasing them into a shelf before an actual request for the machine is received. Currently, only manual triggering is implemented, which means a user can send a request for pre-provisioning a number of machines ahead of expected workload surge, as an example.

Although the core functionality is implemented, discovered limitations regarding monitoring and visibility mean there still is work ahead to continue developing the feature. In the long-term horizon, if proven valuable, multiple other mechanisms to trigger pre-provisioning, as outlined in the Section 4.3.1, may be implemented. These include automated pre-provisioning based on set limits or usage patterns extraction.

Bibliography

[1] *Alembic documentation.* [cit. 2023-04-15]. Available at: https://alembic.sqlalchemy.org/en/latest/.

[2] *Documentation for Artemis provisioning service.* [cit. 2023-04-15]. Available at: https://testing-farm.gitlab.io/artemis/.

[3] *Molten: modern API framework.* [cit. 2023-05-01]. Available at: https://moltenframework.com/index.html.

[4] *Pooling - Wiktionary.* [cit. 2023-05-01]. Available at: https://en.wiktionary.org/wiki/pooling.

[5] *SQLAlchemy 1.4 Documentation.* [cit. 2023-04-10]. Available at: https://docs.sqlalchemy.org/en/14/.

[6] *Testing Farm.* [cit. 2023-05-01]. Available at: https://docs.testing-farm.io/.

[7] Curry, E. Message-Oriented Middleware. In: *Middleware for Communications.* John Wiley & Sons, Ltd, 2004, chap. 1, p. 1–28. DOI: https://doi.org/10.1002/0470862084.ch1. ISBN 9780470862087. Available at: https://onlinelibrary.wiley.com/doi/abs/10.1002/0470862084.ch1.

[8] Davison, B. A Web caching primer. *IEEE Internet Computing.* 2001, vol. 5, no. 4, p. 38–45. DOI: 10.1109/4236.939449.

[9] De, P., Gupta, M., Soni, M. and Thatte, A. Caching VM Instances for Fast VM Provisioning: A Comparative Evaluation. In: Kaklamanis, C., Papatheodorou, T. and Spirakis, P. G., ed. *Euro-Par 2012 Parallel Processing.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, p. 325–336. ISBN 978-3-642-32820-6.

[10] Emeneker, W. and Stanzione, D. C. Efficient virtual machine caching in dynamic virtual clusters. In:. 2007.

[11] Gregg, B. Methodologies. In: *Systems Performance: Enterprise and the Cloud.* 1stth ed. USA: Prentice Hall Press, 2013, chap. 2, p. 21–88. ISBN 0133390098.

[12] Huang, B., Lin, R., Peng, K., Zou, H. and Yang, F. Efficient Service Deployment by Image-Aware VM Allocation Strategy. In: Yin, H., Tang, K., Gao, Y., Klawonn, F., Lee, M. et al., ed. *Intelligent Data Engineering and Automated Learning – IDEAL 2013.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, p. 252–261. ISBN 978-3-642-41278-3.

[13] Kleppmann, M. Encoding and Evolution. In: *Designing Data-Intensive Applications*. O'Reilly Media, Inc., 2017, chap. 4, p. 111–150. ISBN 9781449373320.

[14] Newman, S. Just Enough Microservices. In: *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. O'Reilly Media, 2019, chap. 1, p. 1–32. ISBN 9781492047797. Available at: https://books.google.cz/books?id=ota_DwAAQBAJ.

[15] Sumathi, S. and Esakkirajan, S. *Fundamentals of Relational Database Management Systems*. Springer Berlin Heidelberg, 2007. Studies in Computational Intelligence. ISBN 9783540483977. Available at: https://books.google.cz/books?id=RjnNA0GW0wsC.

[16] Zhu, J., Jiang, Z. and Xiao, Z. Twinkle: A fast resource provisioning mechanism for internet services. In: IEEE. *2011 Proceedings IEEE INFOCOM*. 2011, p. 802–810.

Appendix A

State-Task Diagram

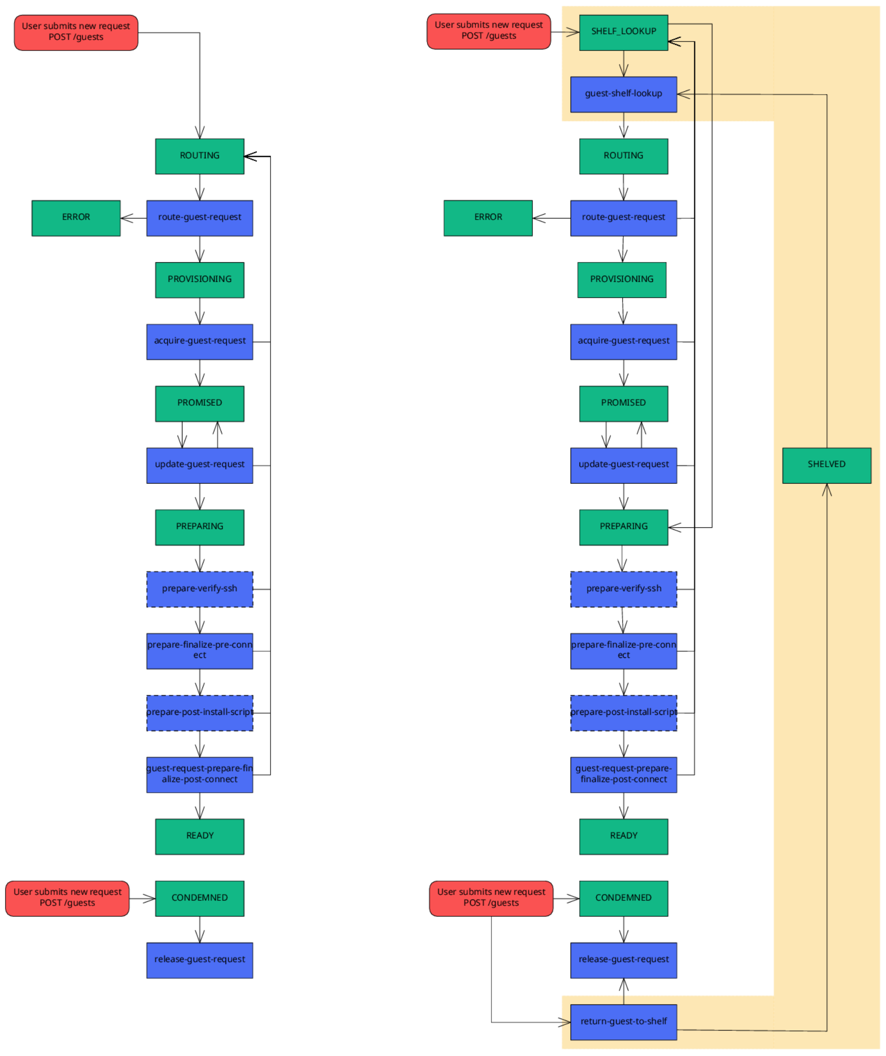

Figure A.1: Side-by-side comparison of diagrams of guest states and the tasks involved in guest provisioning and removal before (left) and after (right) implementing provisioning modifications for shelving

Appendix B

Development Environment

In order to be able to develop, test and verify changes, it is necessary to have a working development environment to run tests and be able to run the Artemis server, as well as interact with the API.

The application is typically deployed to Openshift or Kubernetes container orchestration platforms. This is done using Openshift templates or the application's Helm chart[1].

For local development, the official documentation provides multiple options for deploying the service on local machine [2]. This on the simpler of the two options, running Artemis directly within the installed Python virtual environment.

B.1 Environment Preparation and Server Configuration

Artemis uses Poetry[2] as its dependency manager. It helps to simplify the management of dependencies and ensure consistent dependency versions across different environments. If not present on the development system, Poetry can be installed using the official installer available on the project's website.

After changing the current directory to the server's root,installing Artemis's dependencies is then as simple as running:

```
$ make install
```

To configure the server, it is necessary to create a configuration file `configuration/server.yml` to bootstrap the server. The documentation suggests using the provided templates, however, Artemis contains dummy 'localhost' driver, which is not exposed in this template. This driver does not require access to any infrastructure provider and its use may be desired with the development environment. Therefore, we will proceed with manual creation of the configuration file.

First part can be taken from the original `server.yml.j2` template, extending it by the dummy pool configuration:

```
server.yml:
---

users:
  - name: admin
```

[1]Artemis Helm chart: https://gitlab.com/testing-farm/artemis-helm/

[2]Poetry: https://python-poetry.org/

```
      role: admin

    - name: artemis

  ssh-keys:
    - name: master-key
      owner: artemis
      private: |
        # Your private SSH key (including "BEGIN" and "END" delimiters)
      public: |
        # Public key
      fingerprint: |
        # Key fingerprint

  priority-groups:
    - name: default-priority

  guest_tags: []

  pools:
    - name: local
      driver: localhost
      parameters:
        capabilities:
          supported-architectures: ["x86_64"]
```

B.2 Starting the Server

The server requires access to a database, message broker and a cache. These can be easily started using Docker Compose[3]:

```
$ docker-compose up -d
```

After starting the required services and creating the configuration, the server can be started using the provided script:

```
$ bash nominishift-develop.sh
```

At this point the server should be running and be accessible on `http://localhost:8001`. This can be quickly verified:

```
$ curl http://localhost:8001/about
```

B.3 Configuring CLI

After successfully configuring the server, the CLI can be setup to connect to the local server. There is a command to help guide the user through the configuration of the cli:

[3]Docker Compose: https://docs.docker.com/compose/

```
$ poetry run artemis-cli init
...
---------------------- Artemis API URL ----------------------
URL of Artemis API (for example "http://artemis.example.com/v0.0.18"):
http://localhost:8001/v0.0.56
-------------------- Artemis API version --------------------
API version to use when talking to Artemis (for example "0.0.18"):
0.0.56
...
```

After completing the setup guide, it should be possible to access and list resources available on the server, such as listing the available guests by:

```
$ poetry run artemis-cli guest list
```

Appendix C

Contents of the Attached Media

- `artemis/` — Source code of the Artemis project.
- `thesis/` — LaTeXsource of this thesis along with all presented figures.
- `thesis.pdf` — Thesis in PDF format.
- `tests/` — Source code of the scripts used for evaluation of the results.
- `README.md` — Description of the contents contained on the media and installation instructions for the project.