School of Computer Science
FACULTY OF ENGINEERING AND PHYSICAL SCIENCES



Final Report

LEODIS: Design and Implementation of a Secure, Language-Agnostic, and Self-hostable Web Application Backend

Evan Madurai

Submitted in accordance with the requirements for the degree of BSc Computer Science (Digital & Technology Solutions)

2025/26

COMP3932 Synoptic Project (EPA)

The candidate confirms that the following have been submitted.

Items	Format	Recipient(s) and Date
Final Report	PDF file	Uploaded to Minerva (28/04/2026)
Link to online code repository	URL	Sent to supervisor and assessor (28/04/2026)

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

14 April 2026

(Signature of Student) 

Evan Madurai

© 2025/26 The University of Leeds and Evan Madurai

Summary

This report presents LEODIS, a self-hostable Backend as a Service (BaaS) framework designed to address a gap in the existing landscape of web application backend solutions. Existing solutions force developers to choose between the operational complexity of a traditional three-layer architecture and the vendor lock-in and language restrictions of cloud-hosted BaaS platforms such as Firebase and Supabase. Solutions that achieve single-binary deployment, such as PocketBase, do so without any isolation boundary between application logic and the database, introducing stability risks.

LEODIS fills this gap in the industry by co-locating a document database and an application logic execution environment within a single deployable binary, while maintaining strong isolation through WebAssembly. Stored procedures are compiled to WebAssembly and executed within an isolated environment, ensuring that a buggy or malicious procedure cannot destabilise the host database process. This approach could support stored procedures written in any WebAssembly-compatible language, with Software Development Kits (SDKs) currently provided for Go and Rust.

Performance benchmarks demonstrate that LEODIS achieves significantly lower read latency than both a traditional Gin+MongoDB stack and PocketBase under concurrent load, attributable to its co-location architecture and the use of the Badger storage engine. It however performed poorly in searching workloads due to a lack of secondary indexing and caching strategy. Safety and isolation testing confirmed that fault, resource, and memory isolation guarantees hold in practice.

LEODIS represents a successful proof-of-concept for a new class of web backend architecture, though current limitations include the absence of horizontal scaling and step-by-step procedure debugging.

Acknowledgements

I would first like to thank Professor Arshad Jhumka for his enthusiastic supervision of my project, as well as Dr Chunwei Xia for his advice as my assessor.

I would also particularly like to thank my many coursemates, in particular Eschal Najmi, Morgan Thomas, Rory Edmonds, and George Hester, who have been invaluable sounding boards for ideas, discussions, and rants.

I must also thank my family who have supported me constantly throughout this project and my time at the University of Leeds.

Finally, I would also like to take this opportunity to thank the many people who have helped spark my love for Computer Science. This includes Robert Borland, Tom Scott, Gregor Burns, the Document Distillery team, and the Leeds University Rocketry Association.

Contents

1 Introduction and Background Research **1**

1.1 Introduction . . . 1

1.2 Background Research . . . 2

1.2.1 The architecture of a web application . . . 2

1.2.2 Network access . . . 2

1.2.3 Stored Procedures . . . 3

1.2.4 Execution Isolation . . . 4

1.2.5 Relational and Document Databases . . . 4

1.2.6 Backend as a Service . . . 5

1.3 Literature review . . . 5

1.3.1 FireBase and SupaBase . . . 5

1.3.2 PostgREST . . . 6

1.3.3 SpacetimeDB . . . 6

1.3.4 PocketBase . . . 7

1.3.5 Conclusion . . . 7

1.4 Problem Statement, Project Aims, and Research Questions . . . 8

1.4.1 Problem Statement . . . 8

1.4.2 Project Aims . . . 9

1.4.3 Research Questions . . . 9

2 Methodology **10**

2.1 Requirements . . . 10

2.1.1 Functional Requirements . . . 10

2.1.2 Non-Functional Requirements . . . 10

2.2 Network Protocol . . . 11

2.3 Data model . . . 11

2.4 Secure and isolated stored procedures . . . 11

2.5 LEODIS implementation programming language . . . 12

2.6 LEODIS stored procedure SDKs . . . 12

2.7 Storage Engine . . . 13

2.8 Authentication & Authorisation . . . 14

2.9 Project Management . . . 14

2.10 Evaluation . . . 15

2.10.1 Performance Benchmarking . . . 15

2.10.2 Testing and correctness . . . 16

2.10.3 Security and Isolation Testing . . . 16

3 Implementation and Validation **17**

3.1 System Architecture Overview . . . 17

3.2 Document storage . . . 18
3.3 Software Development Kit design . . . 19
3.4 Stored Procedure handling . . . 20
3.5 Query Processing . . . 20
3.6 Client interface . . . 21
3.7 Concurrency model . . . 22
3.8 Observability . . . 23
3.9 Admin Dashboard . . . 23
3.10 Testing and Validation . . . 23

4 Results, Evaluation and Discussion 25
4.1 Evaluation . . . 25
4.1.1 Safety and Isolation Validation . . . 25
4.1.2 Performance benchmarks . . . 25
4.1.3 Developer Experience . . . 27
4.1.4 Limitations . . . 28
4.2 Future Work . . . 29
4.3 Conclusions . . . 30

References 31

Appendices 35

A Self-appraisal 35
A.1 Critical self-evaluation . . . 35
A.2 Personal reflection and lessons learned . . . 35
A.3 Legal, social, ethical and professional issues . . . 36
A.3.1 Legal issues . . . 36
A.3.2 Social issues . . . 36
A.3.3 Ethical issues . . . 36
A.3.4 Professional issues . . . 37

B External Material 38

C Example code 40

Acronyms

ABI Application Binary Interface. 4, 19, 26

ACID Atomicity, Consistency, Isolation, and Durability. 4, 10, 22

API Application Programming Interface. 5, 15, 17, 18, 20–22

BaaS Backend as a Service. ii, 1, 4, 5, 15

CI/CD Continuous Integration/Continuous Delivery. 27

CRUD Create, Read, Update and Delete. 1, 8, 15

JWT JSON Web Token. 14, 17

LSM Log-Structured Merge-tree. 13, 26

MVCC Multi-Version Concurrency Control. 13, 14, 26

ORM Object-Relational Mapper. 4

REST Representational State Transfer. 3, 20, 27

RPC Remote Procedure Call. 3, 11, 21, 27

SDK Software Development Kit. ii, 1, 5, 10, 14, 16–19, 29

SQL Structured Query Language. 3, 4, 28

UUID Universally Unique Identifier. 18

VU Virtual User. 15

YCSB Yahoo! Cloud Serving Benchmark. 15, 16

Chapter 1

Introduction and Background Research

1.1 Introduction

A significant proportion of real-world software applications can be reduced to Create, Read, Update and Delete (CRUD) operations on a collection of data. Consider software for booking a car rental: there is data (cars available, costs, booking status) and operations performed on that data (adding cars, removing cars, marking a car as booked). Building such an application, however, involves considerable boilerplate work. A seemingly simple application, when developed with the traditional three-layer architecture discussed in 1.2.1, involves configuring a database, building an application layer, and managing authentication before any meaningful application logic can be written [1]. This overhead slows development, raises costs, and increases the possibility of errors made. These costs are a significant financial and productivity burden, a 2024 report found that 69% of developers lost 8 hours or more of their week to inefficiencies and overhead [2].

In response to this perceived complexity, there has been an attempt to consolidate and simplify through the BaaS model, which presents a unified suite of tools such as databases, authentication, and file storage. This allows developers to focus on application logic while leaving configuration, deployment, and infrastructure to the framework and provider [3].

While BaaS solutions serve to reduce development overhead, they introduce their own set of trade-offs. For example these solutions are primarily offered as hosted cloud services, creating a degree of vendor lock-in. Application code is using proprietary SDKs and migrating away from a chosen provider is a significant undertaking [4]. Additionally, the SDKs for each provider typically lock developers into using a particular programming language, reducing developer accessibility to those already familiar with that particular language.

This paper presents LEODIS[1], a self-hostable BaaS framework comprised of a document-style database and an execution environment capable of securely executing application logic through stored procedures written in arbitrary general-purpose programming languages. By co-locating application logic and data storage within a single deployable binary, and executing that logic in an isolated runtime, LEODIS reduces the deployment complexity of a CRUD application to a single unit. A proposed architecture for how LEODIS can be used for developing web applications can be seen in figure 1.1.

Existing BaaS solutions each present a distinct set of trade-offs across seven key parameters: ease of deployment, cloud vendor lock-in, programming language support, authentication granularity, request latency, execution isolation, and scalability. This project occupies a deliberate position within that space, prioritising deployment flexibility and freedom of

[1]Named for the Old English name for Leeds, United Kingdom. Also a backronym for Latency Optimised Document Information System.

language choice over the scalability and ecosystem maturity offered by established cloud providers.

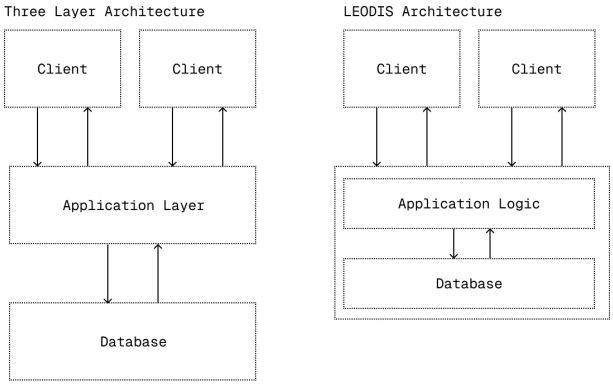


Figure 1.1: Proposal for architecture of LEODIS compared to traditional three-layer architecture

1.2 Background Research

1.2.1 The architecture of a web application

Modern applications are typically built in a three-layer architecture: the user interface, the application server, and the database [5]. In the context of web applications this usually takes the form of the user interface as a web page, the application server as a web framework, and the database as a relational database. Popular examples include MERN (using MongoDB, Express, React, and NodeJS) and LAMP (using Linux, Apache, MySQL, and PHP) [6]. The application server and the database are often referred to as the *backend* of the application. Each of these components are typically separate deployable units, often running in isolated containers or on distinct cloud services. This modular design enables individual scaling, independent deployments, and the substitution of components without modifying the entire system. Each component can be either vertically scaled (increase the available resources on the host machine) or horizontally scaled (distributed across multiple machines to share load). This makes the three-layer architecture extremely scalable and extendible, but more complex to develop and maintain as each unit requires independent configuration, monitoring, and failure handling. Additionally the network boundaries between them introduce both latency and additional failure modes [1].

1.2.2 Network access

While the primary role of the database is storing data, they must also provide an interface through which clients can access and manipulate that data. This is the role of a database wire protocol: a binary protocol that governs how queries are serialised and transmitted to a database server. Common examples include the PostgreSQL wire protocol [7] and the MongoDB wire protocol [8]. A key assumption embedded in these protocols is that the connecting client is a trusted application server rather than an untrusted end user. They do not provide the fine-grained access control demanded by modern web applications [7].

Directly exposing a database over such a protocol to arbitrary clients is dangerous, as any

authenticated client would have broad access to read and modify other clients' data indiscriminately. The conventional solution to this problem is the intermediary application server, which accepts requests from clients, enforces authorisation logic, and translates those requests into database queries [1]. The application server acts as a controlled gateway between the untrusted client and the database. Database wire protocols are typically TCP/IP based, which creates an additional barrier for browser-based clients, as browsers can only communicate over HTTP or WebSockets.

The communication between client and application server is most often done through Representational State Transfer (REST). REST is an architectural style [9] that describes a set of constraints for designing networked systems, most notably statelessness, a uniform interface, and client-server separation. In the real world, REST is usually implemented over HTTP, where the server exposes a set of defined endpoints that the client can interact with using HTTP methods such as GET, POST, PUT, and DELETE, with data exchanged in JSON format. While this combination of REST and JSON has become the convention for web API communication [5], outside of the web there are other methods in use. For example, Remote Procedure Call (RPC) is a paradigm in which a client invokes a named procedure on a remote server as if it were a local function call, abstracting away the underlying network communication [10]. Rather than interacting with a set of resource-oriented endpoints as in REST, the client calls specific named operations directly. This model that can map more directly onto application logic than the HTTP verb and resource model that REST prescribes.

1.2.3 Stored Procedures

Stored procedures are named, pre-compiled collections of database operations stored directly within the database, which can be invoked by applications as a callable unit [11]. Their primary advantage is that they centralise data processing logic within the database layer itself, reducing the number of round-trip network requests between the application and the database server [11].

While it may be tempting to simplify the backend architecture by implementing application logic inside stored procedures (and thus removing the intermediary application server) this comes with significant drawbacks. In relational databases, stored procedures are written in Structured Query Language (SQL) and its variants (e.g. PL/SQL, PL/pgSQL). These are query languages and are not general purpose programming languages. They lack modern programming language features and have no ecosystem of libraries, making business logic difficult to implement and maintain [1]. Code that is stored in databases is also typically difficult to test, observe, and maintain with version control [12].

There are also performance risks with this idea: poorly written stored procedures can introduce bottlenecks, since they execute directly within the database engine without an isolated runtime, a badly optimised procedure can degrade performance for all concurrent users. For these reasons, stored procedures are best suited to well-defined, performance-critical data operations rather than as a general mechanism for encoding application logic.

1.2.4 Execution Isolation

In the context of an execution environment, isolation can be defined as the guarantee that a guest module cannot affect the host process or other guest modules beyond outside its defined interface [13]. This encompasses three distinct properties: *fault isolation* (a crashing procedure must not crash the host), *resource isolation* (a runaway procedure must not continually consume resources), and *memory isolation* (a guest must not read or write host memory outside of the defined Application Binary Interface (ABI) without permission from the host).

In the context of a BaaS attempting to execute application logic within stored procedures, isolation is particularly important as developer-supplied procedures execute alongside trusted host infrastructure. Without an isolation boundary, a buggy or malicious procedure could corrupt other concurrently executing procedures, destabilise the host process, or gain unauthorised access to host resources such as the filesystem and network [13].

The traditional solution to this problem is to place each procedure in its own hardware address space, this allows the operating system to use its memory protection systems to prevent one module from corrupting another. However this hardware level protection comes with prohibitive performance costs for tightly-coupled modules, being as much as two to three orders of magnitude slower than local procedure calls. The modern solution is software-enforced fault isolation. This places modules in a logically separate region of the address space and using code transformation techniques to prevent it from generating addresses outside that region. This is the method used by systems such as NaCl and later WebAssembly [13] [14].

1.2.5 Relational and Document Databases

Databases provide the persistent datastore for modern web applications. They are responsible for storing all application data (e.g. user data). Databases come in a multitude of formats, but the most common for web applications are *relational databases* and *document databases* [1].

Relational Databases

Relational databases are the default approach for web applications, they organise data using a relational modal in which data is stored in tuples with relations between them. This data is then most often retrieved with queries expressed in SQL. Relational databases often come with strict guarantees of Atomicity, Consistency, Isolation, and Durability (ACID) [15], and require data to conform to strict schemas.

Relational databases suffer from the *object-relational impedance mismatch* [16] where application code that typically represents data as objects or JSON structures requires translation to and from the tabular relational model. This is commonly handled with an Object-Relational Mapper (ORM), this however introduces considerable performance overhead, which benchmarks show can be up to 5× slower than raw queries [17].

Document Databases

Document databases store data as self-contained documents, without any external relations. They are often stored in JSON [18] and do not necessarily conform to any given schema,

allowing a more flexible approach to storing data. These became popular in the mid 2000s as the rise of the internet and social media demanded datastores that allowed less structured schemas [19]. They also solve the object-relational impedance mismatch problem since both application and database use JSON for representing data. Popular examples of document databases include MongoDB [20] and DynamoDB [21].

While document databases offer considerable flexibility, this comes with trade-offs when compared to relational databases. Because documents are self-contained, data that would be normalised into separate tables in a relational system is often duplicated across multiple documents, which can increase storage overhead and complicate updates that must be applied consistently in many places [22].

1.2.6 Backend as a Service

The perceived deployment complexity of the three-tier architecture motivated a category of products known as BaaS. Rather than requiring developers to independently configure, deploy, and maintain a database, application server, and authentication system, a BaaS solution packages these into a unified offering. This allows developers to focus on writing application logic and features rather than infrastructure, significantly reducing the time-to-market of web applications [3].

A typical BaaS offering provides services such as a database, authentication, and an automatically generated Application Programming Interface (API) layer as a single integrated unit. The direct client access problem described in the previous section is typically solved through declarative access control policies that are enforced at the database level, removing the need for a hand-written application server to act as a security gateway. These solution allow the implementation of additional application logic through the use of serverless functions [23].

However, BaaS solutions introduce their own trade-offs. Because they are primarily offered as hosted cloud services, the application code becomes tightly coupled to a provider's proprietary SDKs and APIs. Migrating away from a provider is a significant undertaking [4]. This vendor lock-in can be particularly problematic for applications in regulated industries where there is data location requirements [4]. Additionally, most BaaS providers restrict the languages in which application logic can be written, typically supporting only JavaScript, TypeScript, or Python, limiting developer choice and the libraries available for use.

1.3 Literature review

This section of the paper focuses on analysing existing solutions, in particular analysing the trade-offs made in Firebase, Supabase, PostgREST, SpacetimeDB, and PocketBase.

1.3.1 FireBase and SupaBase

While BaaS solutions like FireBase and SupaBase introduce significant benefits such as increased development velocity through reduced infrastructure overhead, they introduce their own trade-offs, most notably vendor lock-in and the restriction in programming language

environment. For example, Firebase can only be used through Google Cloud which could provide issues for regulated industries where data must be stored in compliant locations. While Supabase can technically be self hosted, its missing features such as logging and backups which some developers have found limiting [24]. While these solutions are theoretically scalable, the developer maintains no control over how, when, and where the application is scaled as this is handled by the cloud provider.

Additionally, both FireBase and SupaBase only support the writing of application logic through serverless functions written in Typescript and Python. This limits developer accessibility and narrowly limits the libraries and tools that can be used in development [23].

1.3.2 PostgREST

PostgREST [25] similarly seeks to reduce the boilerplate involved in building CRUD applications. It does so by automatically generating a REST API from a PostgreSQL database schema, exposing each table as an HTTP endpoint with standard GET, POST, PATCH, and DELETE operations [26]. This eliminates the need to hand-write an application server entirely, which leads to a significant reduction in developer effort for simple data-driven applications. This is particularly useful for applications with significant and regular schema changes, as they can be automatically translated to the REST API instead of being done manually, saving developer time.

However, PostgREST still requires the deployment of two separate services, the database and the REST API. They remain a separate process that communicates over the PostgreSQL wire protocol. A deployment still consists of two independently managed services. While both can run on the same host, they are architecturally distinct units with separate configuration, resource footprints, and failure modes. This solutions does not solve the work of deploying and maintaining a three-layer architecture.

A further limitation of PostgREST is its access control functionality. Permissions are expressed entirely through PostgreSQL roles. This system provides a relatively coarse-grained mechanism not particularly suitable for modern web applications. Additionally, it is not possible to extend the functionality of the REST API beyond pure CRUD, additional application logic requires deploying an additional unit.

1.3.3 SpacetimeDB

SpacetimeDB is a backend system developed by Clockwork Labs, released in 2024, which takes a novel approach to the co-location of application logic and data storage. It was designed primarily for the development of massively multiplayer online video games (MMOs) [27], a domain that demands extremely low-latency data access and high write throughput. Rather than exposing a query interface, SpacetimeDB provides network-callable stored procedures as the sole mechanism for client interaction with the database, implemented as WebAssembly (WASM) modules [28]. This allows developers to write application logic in a general-purpose programming language (Rust and C# are currently supported) which is then compiled to WASM and executed within the database process itself.

SpacetimeDB uses a relational data model internally. However, its durability and consistency guarantees are relaxed compared to a traditional database [29]. This is a deliberate design trade-off that is acceptable in video game workloads where losing a fraction of a second of state is often preferable to the latency cost of strict durability guarantees. This would however be inappropriate for many other application domains such as financial or medical systems. It is however a compelling proof of concept for a co-location architecture, particularly with regards to performance [30] but one that is narrowly optimised for its target domain. SpacetimeDB is also limited in scalability, as it only supports vertical scaling without support for horizontal scaling, this places an upper bound on the scalability of SpacetimeDB.

1.3.4 PocketBase

PocketBase is a backend solution that aims to be an open source backend in 1 file. It provides developers with an embedded relational database, ”REST-ish” API, admin dashboard, and authentication [31].

PocketBase is unique compared to other solutions in that it is designed as a Go library rather than a standalone server binary. Developers extend and customise it by writing Go code directly using its library SDK, which is then compiled into the same binary as the core PocketBase runtime. This means the entire backend, including both database and business logic, deploys and runs as a single deployable unit, which simplifies both distribution and deployment [31].

A significant advantage of PocketBase compared to other solutions is that it can be extremely easily self-hosted [32]. This prevents the cloud provider lock-in issues of solutions like Firebase and Supabase, and provides hosting and deployment flexibility.

However, this method of co-location of application logic and database infrastructure carries risk. Poorly written or buggy application code executes within the same process as the database engine, meaning that a crashing or resource-intensive handler can destabilise or degrade the database itself. This tight coupling makes a trade-off for the deployment simplicity of a single binary against the operational isolation that a separate application server would otherwise provide. Since PocketBase is a single binary, it also does not support horizontal scaling, limiting its usecase to applications with predictable and relatively low client load.

1.3.5 Conclusion

There have recently been attempts at simplifying the backend web development process, but each of these approaches carry significant trade-offs that limit their usefulness. Table 1.1 summarises the key trade-offs identified across each solution against the criteria most relevant to LEODIS.

Criterion	Firebase	Supabase	PostgREST	SpacetimeDB	PocketBase	LEODIS
Self-hostable	✗	∼	✓	✓	✓	✓
Single binary	✗	✗	✗	✓	✓	✓
Execution isolation	✓	✓	✗	✓	✗	✓
Language flexibility	✗	✗	N/A	✓	✗	✓
Horizontal scaling	✓	✓	✓	✗	✗	✗
Vendor lock-in	High	Medium	Low	Low	Low	Low
ACID compliance	✓	✓	✓	∼	✓	✓

Table 1.1: Comparison of existing BaaS solutions against LEODIS across key evaluation criteria. ∼ denotes partial support.

As Table 1.1 illustrates, no existing solution simultaneously achieves self-hostability, single-binary deployment, execution isolation, and language flexibility. Solutions that prioritise ease of deployment, such as Firebase and Supabase, do so at the cost of vendor lock-in and language restriction. Solutions that achieve single-binary deployment, such as PocketBase, do so without any isolation boundary between application logic and the database. SpacetimeDB comes closest to the target combination, demonstrating that co-located isolated execution is technically viable, but remains narrowly optimised for video game workloads with durability trade-offs unacceptable for general web application development. This gap, a self-hostable, single-binary backend with isolated, polyglot stored procedures, is the space LEODIS is designed to occupy.

1.4 Problem Statement, Project Aims, and Research Questions

1.4.1 Problem Statement

Building and deploying a CRUD web application today requires a developer to make an important choice between two main options. They can adopt the traditional three-tier architecture, accepting the burden of developing, configuring, deploying, and maintaining multiple independent services, along with the inter-service latency and failure modes that come with them. Or they can adopt a BaaS solution, accepting vendor lock-in, restricted language choice, and loss of control over their infrastructure.

Attempts to find a middle ground, such as PocketBase, achieve deployment simplicity but do so by coupling application logic and database infrastructure in the same process without any isolation boundary. A single poorly implemented procedure can destabilise the entire application. SpacetimeDB demonstrates that application and data co-location with isolated execution is technically viable, but is narrowly designed for video game workloads and makes durability trade-offs that are unacceptable for most general application development.

The result is that a developer who wants a self-hostable single-binary deployment, the freedom to write application logic in the programming language best suited to their problem, and the guarantee that application code cannot destabilise the database it runs alongside currently has

no viable option. This combination of deployment simplicity, language flexibility, and execution isolation is the problem LEODIS is designed to address.

1.4.2 Project Aims

The goal of this project is to design and implement LEODIS, a backend web development framework built around a document style database with network-accessible stored procedures that can be implemented in general-purpose languages and executed safely in isolation. A developer using LEODIS should be able to deploy a complete, production-capable CRUD backend as a single binary with no external dependencies, write their application logic in whichever general-purpose programming language is most appropriate for their use case, and have confidence that a buggy or malicious procedure cannot destabilise the database it runs alongside.

1.4.3 Research Questions

- RQ1: Is it possible to create a stored procedure execution environment that can provide sufficient isolation to mitigate the process-coupling risks present in solutions like PocketBase?
- RQ2: What are the performance characteristics (particularly with regards to latency) of a co-located document database and application logic layer compared to a traditional separated architecture?
- RQ3: How does a single-binary BaaS model compare to traditional three-tier deployment in terms of operational complexity?

Chapter 2

Methodology

2.1 Requirements

The following requirements define the features, expected behaviour, and security considerations of LEODIS' database system and its stored procedure execution environment.

2.1.1 Functional Requirements

- The system shall support the creation, retrieval, updating, and deletion of documents within named collections, with optional persistence to disk.
- The system shall allow developers to define stored procedures in general-purpose programming languages.
- The system shall provide an SDK for writing stored procedures in such a way to abstract any implementation details of the execution environment.
- The system should be deployable as a single binary with no external runtime dependencies.
- Stored procedures should be isolated, yet capable of executing other stored procedures if allowed by the host.
- The system should provide flexibility in authentication and authorisation, making no prescriptions in the shape of user data.
- The system shall be ACID compliant.

2.1.2 Non-Functional Requirements

- The system shall not be dependent on any specific cloud provider.
- The system shall provide memory isolation between the host process and stored procedures such that a crashing procedure does not crash the database.
- The stored procedures shall not be able to access the host filesystem or network.
- Stored procedures shall not be able to access host memory outside of the explicitly provided SDK interface.
- Stored procedures shall not be able to access the memory of other stored procedures without host permission.

2.2 Network Protocol

LEODIS differs from most web application backend frameworks by using an RPC communication paradigm, rather than the REST paradigm that is typically used for web development. To implement this as simply and easily as possible, LEODIS was designed to use the JSON-RPC 2.0 standard [33]. This is a minimal standard that uses JSON as a messaging standard over an RPC communication paradigm, it is agnostic to transport implementation, but as LEODIS is designed as a web application framework it uses HTTPS.

2.3 Data model

LEODIS uses a document database for its data model. Since LEODIS communicates with clients by exchanging JSON over a network, a document-oriented storage model is a natural fit. Data can be stored and retrieved in the same format in which it is transmitted, with no translation layer required. This avoids the object-relational impedance mismatch described previously. Document databases also follow a schema-on-read approach, meaning that the structure of a document is not enforced at the database level but is instead interpreted by the application at read time [1]. This is well suited to the design of LEODIS, where stored procedures written by the developer are responsible for all data access, the procedure itself can enforce whatever structural expectations are appropriate for a given collection, without requiring a rigid schema to be defined and maintained separately. This gives developers the flexibility to iterate on their data model without migrations.

2.4 Secure and isolated stored procedures

One of the requirements of LEODIS is that it provides a sandboxed, isolated, and secure stored procedure execution environment. These stored procedures are the method developers use to extend and add additional application logic to a LEODIS deployment. These can be used to implement authentication logic, read and modify data within the database, or any other bespoke application logic. This is a key differentiator between this project and solutions such as Pocket Base. This was done through the use of WebAssembly.

WebAssembly (often abbreviated to WASM) is an assembly language and virtual machine that is designed to facilitate the embedding of high-performance, sandboxed applications into diverse host environments [14]. While the original goal was to embed application logic into web pages, it has seen a growing use in non-web environments [34]. For example, E-commerce platform Shopify allows stores to add additional functionality via WebAssembly [35] and Figma enables the use of third party plugins that have been compiled to WebAssembly [36].

The main advantage of using WebAssembly for this kind of extensibility is that it language-agnostic. Once compiled to WASM, the host cannot tell the difference between a function written in C++ or Go. This allows developers to work in a programming language and environment they are most comfortable with, increasing development velocity.

The use of WebAssembly for the execution environment would allow developers to implement

their application logic in any compatible language, compile it to WebAssembly, and upload it to a LEODIS instance. The polyglot nature of WebAssembly allows developers to write functions in a language most appropriate. Perhaps execution-time critical applications can be written in Rust taking advantage of its increased performance, while slower more analytic functions could be written in Go. In particular, this project uses Wazero [37] as its WebAssembly runtime due to its fast performance and zero-dependency nature making it easy to integrate into a database environment.

This achieves isolation as the WebAssembly runtime executes each guest module in a separate sandboxed environment. A buggy or poorly performing stored procedure will not destabilise the host process, for example if a guest module panics or enters an infinite loop the Wazero runtime can trap and recover from the error without affecting the database engine or other concurrently executing procedures. Furthermore, WebAssembly modules operate within a strictly bounded linear memory space and have no access to host memory outside of what is explicitly exposed through imported functions [14]. In LEODIS, the only functions exposed to guest modules are those provided by the SDK such as database query and data serialisation utilities. The deny by default security model of WebAssembly means a stored procedure cannot read arbitrary host memory, access the filesystem, or open network connections unless the host explicitly permits it.

2.5 LEODIS implementation programming language

Go was chosen as the programming language for the implementation of LEODIS for the following reasons:

- As a compiled general purpose programming language, Go is highly performant [38] especially compared to interpreted languages.
- Go uses garbage collection, guaranteeing memory safety in non-concurrent workloads [39].
- Go has rich WebAssembly support through Wazero [37], a zero dependency WebAssembly runtime.

Rust was also considered due to its similarly strong, if not better, performance and memory safety guarantees. However, Go was ultimately selected due to greater familiarity with the language, reducing implementation risk and development overhead.

2.6 LEODIS stored procedure SDKs

Since LEODIS executes stored procedures as WebAssembly modules, any programming language that can be compiled to WebAssembly can theoretically be used for writing them. However, each language requires a dedicated SDK to be written before it can be used in practice. This SDK provides the interface between the stored procedure and LEODIS, exposing functions for database interaction and handling the serialisation of data passed between the host (LEODIS) and the guest (the stored procedure module).

Not all languages are equally well-suited to WebAssembly as a compilation target. Since

WebAssembly is a low-level binary instruction format [40], compiled languages are a significantly better fit than interpreted ones. Interpreted languages such as Python or JavaScript rely on a runtime interpreter to execute code, in the context of WebAssembly this requires bundling the interpreter itself into the module, producing large inefficient binaries and reducing performance [41].

Memory management strategy is a further consideration. Languages with a garbage collector, such as Go or Java, must either bundle their garbage collector into the WASM module or depend on garbage collector support from the host runtime. While a WebAssembly garbage collector proposal exists [42], languages with garbage collection can introduce unpredictable pause times that are suboptimal in a latency-critical database context. Languages with compile-time memory management, such as Rust, avoid this entirely and produce smaller and more deterministic modules as a result.

LEODIS currently supports SDKs for two languages. The first, Go, was chosen as the initial target because LEODIS itself is written in Go. To keep WASM module sizes manageable, the SDK uses TinyGo rather than the standard Go compiler. TinyGo is an LLVM-based compiler designed for embedded and WebAssembly targets, and it produces significantly smaller output by reducing the garbage collector overhead that would otherwise bloat a standard Go WASM build. The Go SDK serves as the reference implementation.

The second is Rust, which is probably the most natural fit for WebAssembly of the two. Its ownership-based memory model means there's no garbage collector at all, resulting in small, performant, and deterministic modules.

2.7 Storage Engine

The storage engine is the component of a database responsible for the physical organisation and retrieval of data [43]. LEODIS abstracts the storage layer behind a common interface, allowing the underlying engine to be swapped without affecting the rest of the system.

The decision was made to use an off-the-shelf storage engine to expedite development time. The focus of this project is on the stored procedure model and developing a fast, ACID compatible storage engine, while interesting, are out of scope for this project.

Two storage engines are supported by LEODIS:

In-memory: Data is stored entirely in volatile Random Access Memory (RAM), offering the lowest possible read and write latency since no disk I/O is involved. This is useful for development, testing, and workloads where persistence is not required. However, all data is lost when the process exits, making it unsuitable for production use cases where durability is required.

Badger: A pure Go embedded key-value store developed by Dgraph, inspired by the Log-Structured Merge-tree (LSM) design of WiscKey [44]. Since it is an embedded storage engine, Badger requires no separate database process and fits naturally into LEODIS' single-binary deployment model. Badger also implements Multi-Version

Concurrency Control (MVCC), allowing fully concurrent reads and writes, dramatically increasing performance.

Due to the fact most applications developed with LEODIS will involve persisting user data, Badger is the default storage engine and is recommended for all production deployments. The in-memory engine is provided primarily as a convenience for local development and automated testing, where the overhead of disk I/O is undesirable and data persistence is not a requirement. Bbolt [45], another embedded key-value store similar to Badger but without MVCC, is provided as a legacy option from early in development. While other solutions such as SQLite and LevelDB were considered, Badger was chosen due to its simple API and easy integration due to being written in Go.

2.8 Authentication & Authorisation

A key design motivation for LEODIS is flexibility in authentication and authorisation, the goal was to design a system that makes no prescriptions about how users are defined, stored, or managed. As a result, registration, credential validation, password hashing, and token issuance are all implemented as stored procedures themselves, operating against whatever collection the developer chooses. Authorisation rules such as ownership checks, role-based access, fine-grained permissions are all expressed as plain application logic in stored procedures, meaning they can evolve freely without touching the database.

In order to make this flexibility safe, responsibility is divided between the host and the guest procedures. The system is built on JSON Web Tokens (JWTs), an industry-standard mechanism for cryptographically verifiable claims. It then injects a trusted caller identity into the procedure's execution context. Procedures opt-in to authentication by requesting this context, and may choose to permit anonymous access or deny the request explicitly.

2.9 Project Management

LEODIS was developed using an agile methodology. The focus was on producing functioning prototypes at various stages along the way. This was due to the experimental nature of the project meaning there were significant unknowns at the beginning of development. For example, the exact communication model between WebAssembly modules and the host process, the capabilities of languages being compiled to WebAssembly, and the performance characteristics of off-the-shelf storage engines. These were the targets of initial prototypes, as working examples of these features could act as major de-risking checkpoints for the rest of the project. These early tests validated the use of WebAssembly for the execution environment, but showed that the choice of Bbolt as the storage engine was inappropriate due to its lack of concurrent writes.

The first major development sprint took place in December of 2025, this produced a database capable of executing WebAssembly stored procedures. The second major development sprint took place in March of 2026, this produced more advanced query support, additional programming language SDKs, and an admin dashboard. The tests (described in 2.10.2) were

written alongside features.

The source code was managed using industry standard version control techniques, for which Git and Github were used. This included the use of feature branches, where features were only merged into the main branch once complete and the test suite passed without errors. Merge commits follow the conventional commits [46] standard.

2.10 Evaluation

2.10.1 Performance Benchmarking

Performance benchmarking is focused on end-to-end request latency, this is the total elapsed time between a client sending a request and receiving a response. This metric was chosen because it includes the cost of the full request pipeline (network, query, storage, execution, etc) from the user's perspective. Rather than reporting mean latency, the benchmarks report the 50th (P50), 95th (P95), and 99th (P99) percentiles. The mean is easily skewed by outliers and can present a misleadingly optimistic picture of system performance while percentile metrics give a more honest account of typical and worst-case behaviour under load [47].

To provide a meaningful point of comparison, two equivalent applications were implemented. The first represents the traditional three-layer approach to web development with an application layer written in Go using the Gin HTTP framework and a MongoDB database connected via the official Go MongoDB driver. MongoDB was configured to use the default *read uncommitted* isolation level. This was chosen as it is the default and most commonly used configuration [48]. The second represents a BaaS solution, implemented with PocketBase. This was chosen over Firebase or Supabase as it can be self hosted allowing benchmarks to be completed without introducing internet latency as a confounding variable. Both of these comparison application were designed to expose an identical API to LEODIS, meaning the same benchmarking scripts can be used against both systems without modification.

Benchmarking was performed using k6, an open-source load testing tool. Tests were run locally on the same machine for all applications. All results should therefore be interpreted as measures of relative performance between the two architectures rather than figures from a production environment. The hardware used for benchmarking was an M4 Macbook Pro with 24GB of unified memory. The key parameters varied throughout the test are Virtual User (VU)s and the quantity of documents in the database during the benchmark. Every test was ran for an equal 60 seconds to gain insights under sustained loads. The database was purged before every test and re-populated with consistent generated sample data.

The performance benchmarks were designed in the spirit of the Yahoo! Cloud Serving Benchmark (YCSB) [49], a widely adopted framework for evaluating the performance of cloud data serving systems through standardised, application-motivated workloads. Rather than adopting YCSB directly, the benchmarks were adapted to reflect the specific access patterns of CRUD web applications and the architectural characteristics of LEODIS. The following benchmarks were therefore used:

`doc_list`: A concurrent read benchmark which involves reading the entire contents of the

database. This is analogous to *YCSB Workload C* and was designed to emulate a large read request such as listing all available users in a system.

`doc_update`: A concurrent mixed read/write benchmark consisting of 50% read operations and 50% write operations, with documents selected according to a Zipfian distribution to reflect the realistic access patterns of web applications, where a small number of records receive a disproportionate share of requests [49]. This is analogous to *YCSB Workload A* and was designed to evaluate LEODIS under a realistic workload.

`doc_insert`: A highly concurrent write benchmark which involves thousands of simulated users attempting singular writes to the database. This was designed to emulate the constant stream of small writes present in applications such as e-commerce and social media.

2.10.2 Testing and correctness

Software tests are often divided into three categories: unit tests, integration tests, and end-to-end tests. The correct balance of which tests to use is a subject of disagreement, the more traditional approach is to focus on unit tests [50] as there was significant cost and time investments required for end-to-end tests. However some argue that with recent advancements in testing technology, end-to-end tests can now provide the most utility for developers without the cost and time trade-offs of prior years [51].

The testing suite for LEODIS mostly focuses on unit tests, this was chosen due to the nature of the system being tested. Since LEODIS is a framework rather than a user-facing application, it's public interface is its components. Unit tests are well suited to verifying the correctness of the components that make up LEODIS such as the storage engine, the WASM execution environment, the JSON-RPC request parsing, and the SDK interfaces. Integration and end-to-end tests were de-prioritised as the primary consumers of LEODIS are developers writing stored procedures rather than end users interacting with a fixed interface.

2.10.3 Security and Isolation Testing

In order to correctly verify the safety and isolation of the stored procedures, a series of more in-depth tests were implemented that involve executing problematic stored procedures. These were designed to be deliberately slow, badly written, or insecure, with the goal being to test the guarantees that WASM claims to provide.

These properties are theoretically provided by the WebAssembly specification itself [14], but the correctness of LEODIS must be independently validated as it depends on how the WASM runtime (Wazero) is configured as well as how the SDK is implemented. The tests described here are therefore designed to validate LEODIS' specific implementation.

Chapter 3

Implementation and Validation

3.1 System Architecture Overview

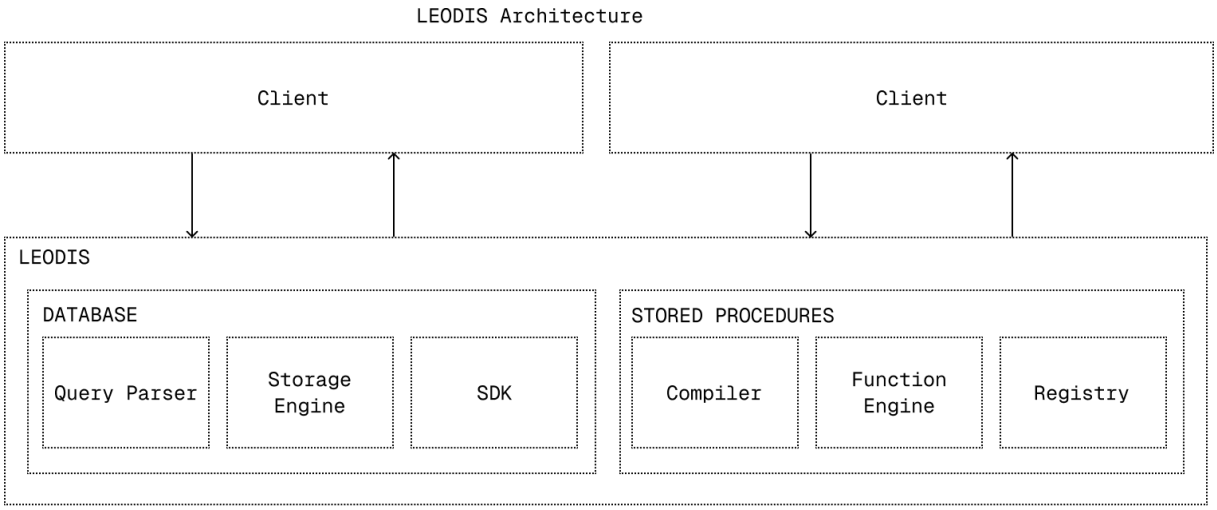


Figure 3.1: LEODIS Component Architecture

As seen in figure 3.1, LEODIS was implemented in a highly modular fashion with the codebase being highly separated between interface and implementation. The following components comprise the core of LEODIS and their corresponding role.

- Network Handler: Processes incoming HTTP requests.
- Authenticator: Handles the signing and validation of JWT tokens to authenticate incoming requests.
- Observability: Handles logging and event tracking.
- Registry: Contains all metadata for which stored procedures are loaded into the database.
- Compiler: Handles the compilation of stored procedures into WebAssembly.
- SDK: The SDKs provide the stored procedures with an API to interact with the database.
- Function Engine: Handles the loading, executing, and monitoring of stored procedure modules.
- Query Parser: Handles the parsing and processing of queries to retrieve data from storage engine.
- Storage Engine: Handles the persistence of data to disk.

A common workflow of LEODIS, as seen in figure 3.2, starts with a client requesting that LEODIS execute a stored procedure. The client can use its client library to request execution of a stored procedure on the database, this will be received by the `network handler` which will process the incoming HTTP JSON-RPC request. This request will then be handed to the Authenticator for verifying a valid authentication token is present on the request, if true, a flag is attached to the request to confirm it has originated from an authenticated client. The `function engine` is then invoked with the requested stored procedure. This stored procedure will likely need to request data from the database through the SDK. The SDK invokes the `query parser` which processes the query into its component parts. These parts can then be executed against the `storage engine` to retrieve the data from disk. This is then passed back up the chain back to the client.

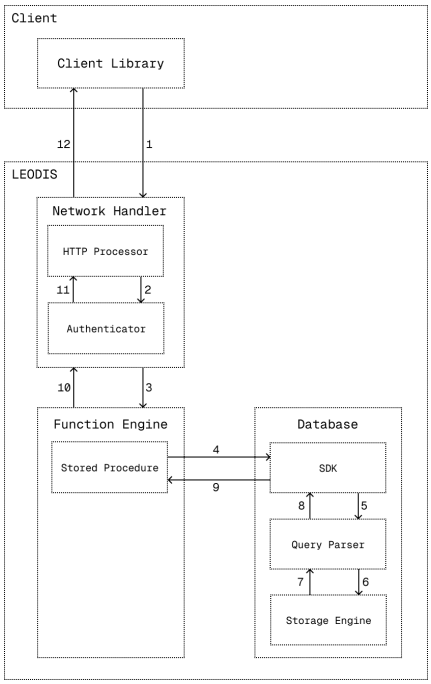


Figure 3.2: Data flow during typical LEODIS invocation.

3.2 Document storage

To ensure the persistent storage of documents, LEODIS uses a thin API over an abstracted key–value store, with methods such as `Put`, `Get`, `Delete`, and `ListKeys` for adding, reading, and deleting documents from the database, respectively. Each document is stored in a `collection` (analogous to a table in a relational database) as a single row in the key–value store. An example of a collection can be seen in figure 3.3. The serialisation and deserialisation of data to and from disk is handled by the storage engine. As LEODIS' database is a document style database, there is no enforced schema on a document or collection level, with the only enforced field being a Universally Unique Identifier (UUID).

One limitation to the document model as implemented is that queries are not index-driven at the key-value store level. The `Find` method involves a pass over every key in the collection with no secondary indexing. This is a performance bottleneck as it results in searches being $\mathcal{O}(n)$, causing potentially high search times in large collections.

```
{
  "documents": [
    {
      "_id": "2281a920-a8d2-4da8-ad4f-b9f6e0065ac8",
      "name": "Stripes",
      "age": 11,
      "weight": 6
    },
    {
      "_id": "bec38134-95e5-4a2a-a18b-9c2695ba91e4",
      "name": "Jasmine",
      "age": 11,
      "weight": 4.2
    }
  ]
}
```


Figure 3.3: Example of a collection stored in LEODIS' database.

3.3 Software Development Kit design

In order to enable development of stored procedures, a library is needed to facilitate communication between the guest stored procedure and the LEODIS host. This is provided by the SDK that is responsible for all serialisation and communication across this boundary.

WebAssembly's native type system permits only integer exchange between host (LEODIS) and guest (stored procedure), making it necessary to build a higher-level ABI on top of this primitive foundation. LEODIS achieves this through a series of exported functions exposed to every guest module. This includes database operations for searching, inserting, and deleting documents. Each guest module in turn exports three symbols: `run`, `allocate`, and `deallocate` which give the host safe, controlled access to the guest's linear memory.

All data exchange is conducted via pointer and length pairs into the guest's single linear memory region. The SDK passes strings and JSON by placing them in allocated memory and then passing their address and length to the host, which copies bytes out using `Memory().Read`. For operations that return data the host calls the guest's `allocate` to reserve space and writes the result back with `Memory().Write`. The run completes by returning a `uint64` containing the pointer and length, which is used by the host to read the final JSON document array out of guest memory. End-to-end, stored procedures interact with the database exclusively through this JSON-over-linear-memory ABI.

The SDK is designed to abstract away as much of this memory management as possible, exposing functions in a near-native manner. For example, database searches can be made from the stored procedures through `sdk.Find(collection, query)`, insertions with `sdk.Insert(collection, document)`, and deletions with `sdk.Delete(collection, match)`. Other features provided by the SDK include `sdk.GetInput()` which pipes any provided arguments into the stored procedure and `sdk.Log()` which provides a method of logging from

inside a procedure.

3.4 Stored Procedure handling

In order to manage its stored procedures, LEODIS uses a function registry called `registry.yaml` (see Fig 3.5) that contains a record of all functions loaded into LEODIS. The registry points to each function's `module.yaml` (see Fig 3.4) which contains metadata such as name, tags, and example inputs and outputs. This registry can be thought of as defining the communication contract between LEODIS and its clients, analogous to an OpenAPI spec in REST web application backends.

On startup LEODIS checks this function registry and compiles the source code for each stored procedure to WebAssembly and loads it into the `function engine` component, responsible for the tracking and executing of all stored procedures.

```
id: cat_age
language: go
source: cat_age.go

description: Compute the average age of all cats.
tags:
  - query
metadata:
  example_call: cat_age
  example_input: ""
  example_output: |
    {"documents":[{"average_age":10.5}]}
```

Figure 3.4: Example of a module.yaml.

```
functions:
  - module: modules/cat_list/module.yaml
  - module: modules/cat_age/module.yaml
  - module: modules/cat_insert/module.yaml
```

Figure 3.5: Example of a registry.yaml.

3.5 Query Processing

The stored procedures interact with the database component through a query API modelled after similar databases such as MongoDB and Cosmos DB, making it familiar to experienced developers. It is however much simpler in design. For example, it lacks advanced features such as aggregation pipelines. Developers can use this API for writing database queries within stored procedures.

```
q := sdk.Query{
        Filter: sdk.Or(
                sdk.Gt("age", 10),
                sdk.And(
                        sdk.Gt("weight", 2),
                        sdk.Gt("age", 5),
                ),
        ),
        Sort:  sdk.Sort{Field: "age", Order: 1},
        Limit: -1,
}

packed, err := sdk.Find("cats", q)
```

Figure 3.6: Example of a query from a stored procedure to database component.

For example, the query in figure 3.6 demonstrates an equivalent query for *find all cats where either age is greater than 10* ***or*** *weight is greater than 2 and age is greater than 5, sorted by age ascending and with no limit on number of results.*

3.6 Client interface

As previously established, LEODIS exposes an RPC-based HTTP API to its clients, specifically one conforming to the JSON-RPC 2.0 specification. However, direct use of this API is not the intended method of interaction. Instead, LEODIS is distributed with client libraries that abstract the underlying protocol into idiomatic, high-level interfaces.

Two client libraries are currently implemented: Python and TypeScript. The Python library is intended primarily for scripting and automation use cases, while the TypeScript library is designed for use within web applications. Both libraries encapsulate the JSON-RPC 2.0 wire format entirely, presenting clients with a clean API that requires no knowledge of the underlying transport. An example of how the Typescript client can be used for interacting with a LEODIS instance an be seen in 3.7.

```
import { useMemo, useState } from 'react'
import { Leodis } from 'leodis-client'

function CatApp() {
  const [baseUrl] = useState('https://leodis.evanmadurai.co.uk')
  const [token, setToken] = useState<string | null>(null)

  const client = useMemo(
    () =>
      new Leodis({
        baseUrl,
        getAccessToken: () => token,
      }),
    [baseUrl, token],
  )

  async function loadCats() {
    const { documents } = await client.execute<Record<string, unknown>>(
      'cat_list',
      null,
    )
    return documents
  }
}
```

Figure 3.7: Example of Typescript client library usage

3.7 Concurrency model

As a web backend framework, LEODIS is designed with high performance as a requirement, defined as the ability to process concurrent requests from a large number of simultaneous users. To meet this requirement, concurrency is managed at every layer of the system using mutual exclusion to prevent race conditions.

At the network layer, LEODIS uses the Gin framework [52] to handle HTTP connections, this distributes incoming requests across multiple threads ensuring that no single request blocks the processing of others. Within the database layer, the top-level API protects its internal collection map with a `sync.RWMutex`, permitting multiple concurrent readers while serialising write operations that modify collection metadata. A similar strategy is applied to the WebAssembly function registry: read locks are acquired for registry lookups, while registration and unregistration operations require an exclusive write lock. Notably, the read lock is released prior to executing guest WebAssembly code, allowing multiple WebAssembly functions to execute concurrently without contention on the registry itself. Together, these mechanisms ensure that LEODIS remains safe and ACID compliant under concurrent access at each layer of its execution model.

3.8 Observability

Observability refers to the degree to which the internal state of a system can be inferred from its external outputs such as logs and performance metrics [53]. Observability is a key requirement for LEODIS and as such it has a simple yet relatively robust implementation. Logs can be created from both LEODIS and its stored procedures and events are automatically emitted for operations such as registering or executing stored procedures. These logs are persistent, being stored in a special append-only log within the storage engine. Developers using LEODIS can use these logs and events to diagnose errors within their application.

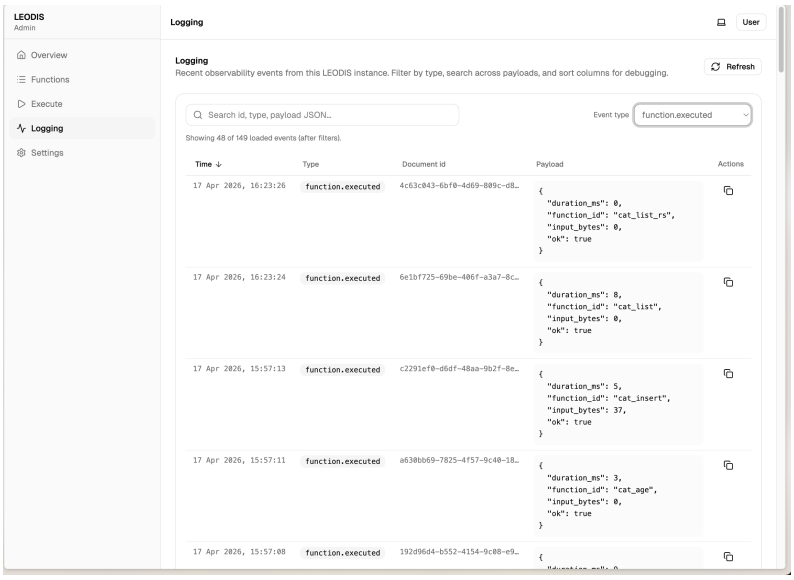
3.9 Admin Dashboard

An additional optional feature of a LEODIS deployment is the admin dashboard. This is a web user interface that allows administrators to view, monitor, and interact with the backend. Figure 3.8 illustrates the three main panels of the admin dashboard: the logging view, available stored procedures, and query execution interface.

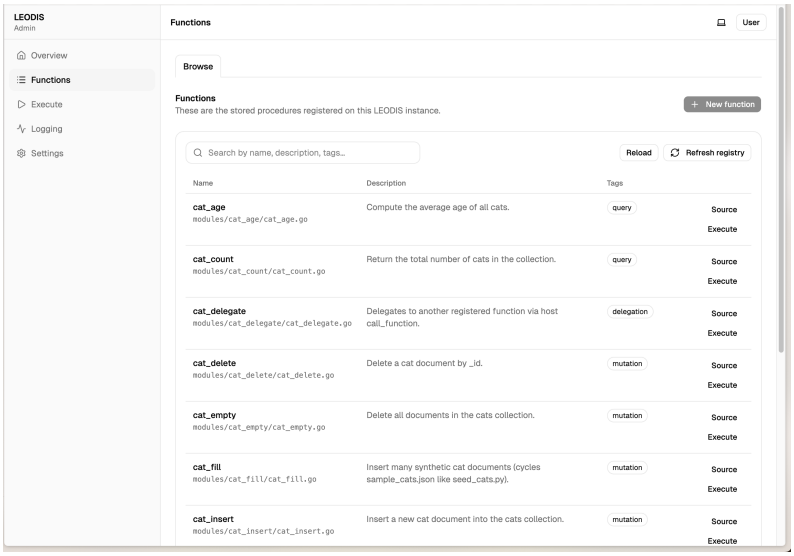
3.10 Testing and Validation

The testing of LEODIS is performed using a suite of 106 unit tests. These cover all individual components of LEODIS. As mentioned previously, a series of in-depth tests to validate the safety and integrity of the WebAssembly stored procedure model were created, these include:

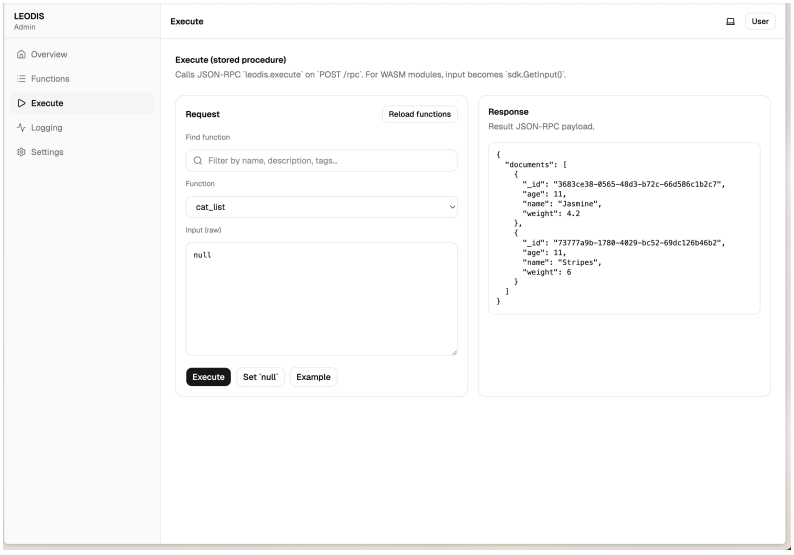
- `TestWasmIsolation_guestPanicHostSurvives`: Designed to test fault isolation. In the event of a stored procedure panic, the host should catch this and continue executing other procedures without any issues. This was implemented by calling `panic.wasm`, a stored procedure that simply immediately returns a panic, and then executing a different procedure after and validating that was executed.
- `TestWasmIsolation_infiniteLoopContextCancels`: Designed to test *resource isolation*. This validates that when a stored procedure continues executing past its timed deadline it is correctly cancelled. This is done by calling `loop.wasm`, a stored procedure that simply infinitely loops without exiting, followed by a standard stored procedure. If the second stored procedure executes, then we know the infinitely looping procedure was cancelled before exit.
- `TestWasmIsolation_invalidResultPointerRejected`: Designed to test *memory isolation*. Validates that when a stored procedure returns out-of-bounds memory addresses, the host safely handles this without crashing. This calls `badresult.wasm`, which deliberately returns some invalid pointer addresses to the host. If the host correctly identifies this and refuses to read from unsafe memory, it has handled this safely.
- `TestWasmIsolation_filesystemDeniedByDefault` and `TestWasmIsolation_networkDeniedByDefault`: Confirms that stored procedures cannot access the host filesystem or network without permission, validating the deny-by-default security paradigm.



(a) Logging view, displaying real-time system and query logs.



(b) Functions panel, listing available stored procedures.



(c) Execution panel, allowing administrators to run queries directly.

Figure 3.8: The LEODIS admin dashboard, showing the logging view, functions panel, and query execution interface.

Chapter 4

Results, Evaluation and Discussion

4.1 Evaluation

4.1.1 Safety and Isolation Validation

The results from the safety and isolation tests, described in Section 3.10, confirm that LEODIS has sufficient execution isolation to mitigate the co-location risks present in solutions like PocketBase. In every case tested, the host process continued operating correctly after the adversarial guest module attempted to perform unsafe or malicious operations. A panicking stored procedure was caught and recovered from without affecting concurrent execution, a runaway procedure was cancelled cleanly when its deadline elapsed, and out-of-bounds memory addresses returned by a malicious module were rejected without a host crash. Filesystem and network access attempts from guest modules were silently denied by default, confirming that the deny-by-default capability model of WebAssembly is correctly enforced in LEODIS' implementation.

4.1.2 Performance benchmarks

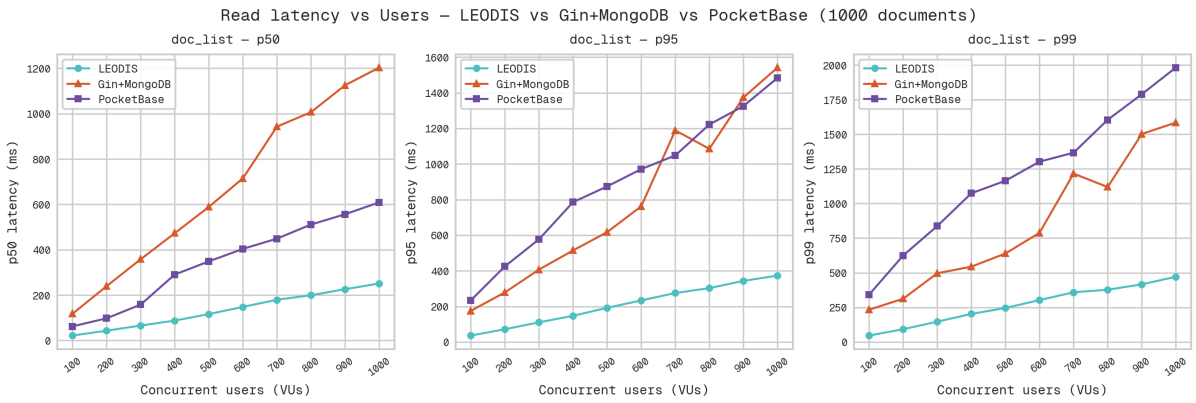


Figure 4.1: Read latency dependence on concurrent users for LEODIS, Gin+MongoDB and PocketBase.

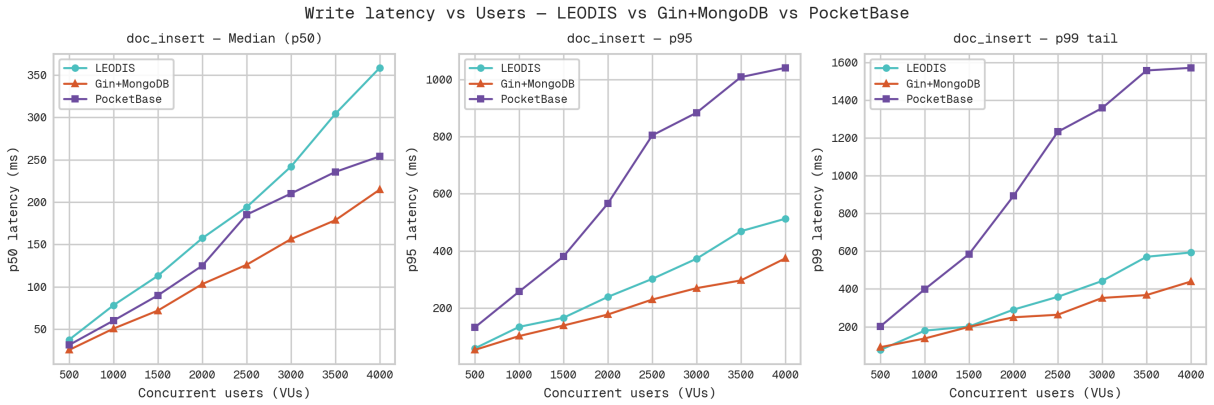


Figure 4.2: Write latency dependence on concurrent users for LEODIS, Gin+MongoDB and PocketBase

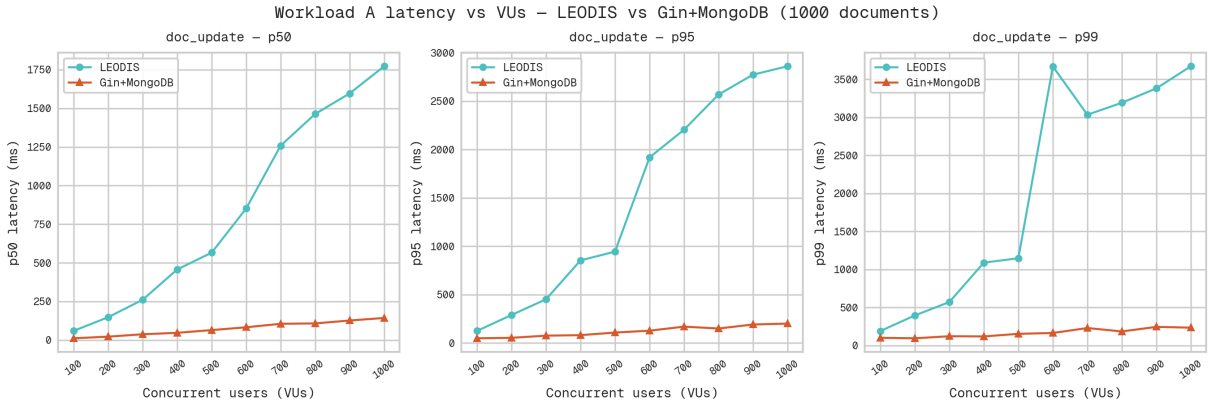


Figure 4.3: Mixed read and update workload request latency, LEODIS and Gin+MongoDB.

In performance benchmarks, as seen in figure 4.1, LEODIS showed a significant strength in concurrent large read workloads. In figure 4.1 it was able to read and return the entire contents of a database collection significantly faster than both the Gin+MongoDB solution and the `PocketBase` solution at each simulated load that was tested. For example, at 1000 concurrent users, LEODIS achieved a P50 latency of 251.04 ms, 2.4× faster than PocketBase and 4.8× faster than Gin+MongoDB. This is likely an advantage of LEODIS' co-location architecture, as it avoids invoking the networking stack at all during execution, instead using an ABI that relies on shared memory. Additionally, the use of Badger as the storage engine likely contributes to this performance as its LSM architecture is optimised for fast reads.

In highly-concurrent write workloads LEODIS shows comparable, if slightly slower, performance to the other tested solutions. As seen in figure 4.2, it is comparable to both other tested solutions in p50 latency. This is likely attributable to the use of Badger as the storage engine, as its MVCC architecture enables concurrent writes. PocketBase struggled consistently in tail latency, performing 3× worse than Gin+MongoDB and 2.8× worse than LEODIS in P95 latency and similarly poorly in P99 latency.

In `doc_update`, as seen in figure 4.3, LEODIS performed significantly slower than the comparable Gin+MongoDB solution. This is likely due to the fact MongoDB features an extremely mature caching strategy, which LEODIS lacks. Since the test disproportionally

reads from certain documents, MongoDB is able to use its cache to dramatically improve performance. Another likely cause is LEODIS requiring an $\mathcal{O}(n)$ collection scan for each search, this is significantly slower than the secondary indexing strategies used by mature databases like MongoDB. This benchmark is missing results from PocketBase as it was unable to complete the benchmark due to repeated crashes from network thread pooling errors. While the root cause of this crash is unknown, it could have potentially been caused by the lack of execution isolation identified earlier in this report, lending credence to the idea this causes instability.

One potential caveat with these benchmarking results is that since PocketBase is designed to use a REST paradigm, where some adaptor code is required to expose a RPC interface. This could potentially introduce some non-representative overhead as there is an additional computational step to translate RPC to REST that would not exist in a typical PocketBase application.

With regards to 1.4.3, the benchmarking results lend credibility to the idea that a co-located architecture can offer meaningful latency advantages over a traditional separated architecture, particularly under concurrent read-heavy workloads. However, this advantage is not universal. The absence of secondary indexing and a mature caching layer means LEODIS underperforms in mixed read/write workloads compared to established databases like MongoDB.

4.1.3 Developer Experience

Deployment and management

A strength of LEODIS is its ease of deployment. Traditional three-layer deployments, such as the Gin+MongoDB application used for benchmarking, require individually configuring a database instance with connection strings, an application server with environment variables, and a reverse proxy such as Nginx. By contrast, LEODIS' single-binary model requires no additional runtimes or external dependencies. The only configuration required is defining the stored procedures themselves. This directly addresses one of the core motivations of the project: reducing the operational overhead of deploying a production-capable CRUD backend.

Stored procedures are managed through declarative configuration files (`registry.yaml` and `module.yaml`) which describe all metadata, expected inputs, and outputs for each procedure. Unlike Firebase, where backend configuration is managed through a web console and is therefore difficult to audit or reproduce, this file-based approach means the entire backend configuration lives in the repository alongside application code. It is deterministic, reviewable in pull requests, and deployable through standard Continuous Integration/Continuous Delivery (CI/CD) pipelines without additional tooling.

Ease of development

When developing stored procedures for LEODIS, the SDK abstracts away the complexity of the WebAssembly ABI entirely. The details of linear memory management, pointer passing, and serialisation are handled by LEODIS, meaning a developer writing a stored procedure interacts exclusively with high-level functions such as `sdk.Find()`, `sdk.Insert()`, and

`sdk.GetInput()`. The query API is deliberately modelled after MongoDB, meaning developers with prior document database experience will find the interface familiar. The polyglot nature of the platform means developers are free to choose the most appropriate available language for a given procedure rather than being constrained to a single runtime as with Firebase or Supabase.

As a result of WebAssembly modules being platform-agnostic by design, stored procedures can be compiled on a developer's local machine and uploaded to a LEODIS deployment regardless of operating system or architecture, eliminating an entire class of environment-specific bugs. The use of WebAssembly also allows the hot-reloading of functions without redeploying LEODIS.

Debugging of stored procedures

One of the identified issues with stored procedures in traditional relational databases was the inability to debug the execution. Despite this being cited as a major motivation for general purpose stored procedures, step-by-step debugging inside a stored procedure is not currently possible in LEODIS. Traditional debuggers attach to a running process and allow developers to inspect state at arbitrary points of execution, this capability does not extend naturally into the WebAssembly sandbox. Developers are therefore reliant on `sdk.Log()` and the admin dashboard for diagnosing procedure behaviour, which is a less powerful tool than a full debugger for complex bugs, but still much more effective than tools for query languages like SQL. This is a considerable downside with using WebAssembly for procedure isolation at present, however, a proposal for better debugging support was recently accepted into the specification [54].

Communication paradigm

Finally, the JSON-RPC communication paradigm may present a learning curve for developers with a strong REST background. While this maps more naturally onto application logic in many cases, it represents a departure from the standard paradigm that most web developers will be familiar with.

4.1.4 Limitations

While LEODIS offers a unique set of features focused on execution isolation, programming language flexibility, and ease of deployment, this comes with downsides worth considering.

A fundamental architectural limitation of LEODIS is its inability to scale horizontally. LEODIS' co-location of application logic and data storage prevents this as the stored procedures execute within the same process as the database, there is no application tier that can be replicated. Running multiple LEODIS instances would require data to be partitioned or replicated across them, introducing the distributed systems problems of consistency and coordination. As a result, LEODIS is limited to vertical scaling, it is only capable of processing greater loads by provisioning a more powerful single machine. This places a hard upper bound on its capacity. This is a trade-off shared by PocketBase and SpacetimeDB, and

is arguably the price of deployment simplicity. LEODIS is therefore best suited to applications with moderate and predictable load rather than those anticipating significant growth.

While LEODIS offers greater programming language flexibility than any comparable solution, this flexibility is not unconditional. The use of WebAssembly imposes its own constraints. Languages and libraries that depend on native C extensions cannot be ported as these are platform specific and usually lack WebAssembly targets. This was encountered during development when attempting to support Python stored procedures. Python itself can be compiled to WebAssembly via projects such as Pyodide, however popular analytical libraries such as NumPy and Pandas are implemented in C and would require significant effort to port to a WebAssembly environment. With additional engineering investment, this could enable a class of analytical focused stored procedures with access to Python's rich data science ecosystem, but the engineering investment required was out of scope for this project. The implication is that LEODIS' language flexibility is most effectively used with compiled languages such as Rust and Go rather than interpreted languages.

4.2 Future Work

A significant limitation of LEODIS identified in this project is its inability to scale horizontally, and addressing this remains an interesting direction for future work. Convex, another backend platform, handles this by ensuring that every node in a cluster executes the same stored procedures against a replicated log, keeping state consistent across machines. However this approach assumes strict requirements on stored procedure behaviour, for example they must be strictly deterministic (including system calls like timestamps) so that identical inputs always produce identical outputs on every node [1]. Extending LEODIS to support this model is possible with significant work.

The set of supported stored procedure languages could be expanded by implementing additional SDKs. The two currently supported languages (Go and Rust) cover the most natural WebAssembly compilation targets, but strong candidates for future SDKs include C and C++, which have mature WebAssembly toolchains, as well as C#, Zig, and Java.

The client libraries currently implemented for Python and TypeScript are intentionally minimal, designed to demonstrate the JSON-RPC interface rather than serve as production-grade tools. Future work could develop these into fully featured libraries with better error handling, retry logic, connection pooling, and type-safe generated clients derived automatically from the `module.yaml` metadata. The latter would be particularly valuable for TypeScript users, where generating typed procedure call signatures from the registry would provide compile-time safety for client code.

LEODIS currently has no support for real-time data subscriptions. Firebase's listener model, where clients receive pushed updates when data they are subscribed to changes, is one of its most widely used features and a capability gap for LEODIS. Adding WebSocket-based subscriptions that emit events when a collection is modified would significantly broaden the range of applications LEODIS is suited to, particularly for collaborative or live-updating interfaces.

Finally, the current storage engine performs collection scans in $\mathcal{O}(n)$ time as there is no secondary indexing. For small collections this is acceptable, but it becomes a meaningful bottleneck as collection size grows, as demonstrated in the performance benchmarks. Adding support for secondary indices on user-defined fields within the storage engine backend would bring query performance in line with production document databases and remove one of the more significant gaps between LEODIS and established solutions like MongoDB.

4.3 Conclusions

This project set out to fill a gap in the industry: to create a self-hostable, single-binary backend framework for CRUD web applications where data and application logic can be co-located with an isolated execution environment. When judged against this criteria, it has been a success. To revisit the research questions presented in 1.4.3:

- **RQ1:** It is possible to create a stored procedure execution environment that provides sufficient isolation to mitigate the process-coupling risks present in other solutions such as PocketBase. Safety and isolation testing confirmed that fault, resource, and memory isolation guarantees hold in practice: panicking procedures were caught without affecting the host, runaway procedures were cancelled on deadline, and out-of-bounds memory accesses were safely rejected. The deny-by-default capability model of WebAssembly was correctly enforced throughout.
- **RQ2:** A co-located architecture can offer meaningful latency advantages over a traditional separated architecture, particularly under concurrent read-heavy workloads, achieving up to 4.8× lower p50 latency than Gin+MongoDB at 1000 concurrent users. However, this advantage is not universal to all workloads. The absence of secondary indexing and a mature caching layer means LEODIS severely underperforms in mixed read/write workloads compared to established databases like MongoDB. This is however not inherent to the architecture and could be improved with further development.
- **RQ3:** The single-binary deployment model comes with a significant reduction in operational complexity compared to the traditional three-layer approach. Where a conventional deployment requires independently configuring a database, application server, and reverse proxy, LEODIS requires no external dependencies or runtimes, only the definition of stored procedures themselves. Configuration is file-based, version-controllable, and deployable through standard CI/CD pipelines without additional tooling.

Taken together, these results demonstrate that the core architectural thesis of LEODIS is valid. Co-locating data storage and application logic within an isolated execution environment is not only technically viable, but offers tangible benefits in deployment simplicity and performance. While not a production-grade solution, LEODIS is a successful proof-of-concept for a new method of web application backend development.

References

[1] Martin Kleppmann et al. Designing data-intensive applications, 2019.

[2] Atlassian, DX, and Wakefield Research. State of developer experience report 2024. Technical report, Atlassian, 2024. Accessed: 2026-04-25.

[3] Ram Kesavan, David Gay, Daniel Thevessen, Jimit Shah, and C Mohan. Firestore: The nosql serverless database for the application developer. In *2023 IEEE 39th international conference on data engineering (ICDE)*, pages 3376–3388. IEEE, 2023.

[4] Justice Opara-Martins, Reza Sahandi, and Feng Tian. Critical analysis of vendor lock-in and its impact on cloud computing migration: a business perspective. *Journal of Cloud Computing*, 5(1):4, 2016.

[5] Maarten van Steen and Andrew S. Tanenbaum. *Distributed Systems*. distributed-systems.net, 4th edition, 2023.

[6] Fatima zahra Iguenfer, Mohammed Benhayoun, and Falah Bouchaib. An overview of mean, mern, and lamp stacks: A comparative analysis of architecture, cost, learning curve, and community support.

[7] The PostgreSQL Global Development Group. PostgreSQL documentation: Frontend/backend protocol, 2026.

[8] MongoDB, Inc. MongoDB wire protocol — MongoDB manual. `https://www.mongodb.com/docs/manual/reference/mongodb-wire-protocol/`, 2026. Accessed: 2026-04-06.

[9] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000.

[10] Andrew D Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, 2(1):39–59, 1984.

[11] Tatyana N Filimonenkova. Stored procedures as an implementation tool business logic in applications of databases. *CEUR-WS. org*, 2522, 2019.

[12] Surabhi Gupta and Karthik Ramachandra. Procedural extensions of sql: Understanding their usage in the wild. *Proceedings of the VLDB Endowment*, 14(8):1378–1391, 2021.

[13] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. Efficient software-based fault isolation. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 203–216, 1993.

[14] WebAssembly Community Group. Introduction — WebAssembly 2.0 core specification, 2024. Accessed: 2026-04-14.

[15] Jim Gray et al. The transaction concept: Virtues and limitations. In *VLDB*, volume 81, pages 144–154, 1981.

[16] Christopher Ireland, David Bowers, Michael Newton, and Kevin Waugh. A classification of object-relational impedance mismatch. In *2009 First International Confernce on Advances in Databases, Knowledge, and Data Applications*, pages 36–43. IEEE, 2009.

[17] Joseph Christian Yusmita, Ronald Arya, Jason Manuel Wijaya, Kristien Margi Suryaningrum, and Ricky Reynardo Siswanto. Optimizing database access strategy: A performance analysis comparison of raw sql and prisma orm. *Procedia Computer Science*, 269:1201–1210, 2025. The 10th International Conference on Computer Science and Computational Intelligence 2025.

[18] MongoDB, Inc. JSON and BSON data types: What is the difference?, 2026.

[19] Keith D. Foote. A brief history of non-relational databases. Dataversity, June 2018. Accessed: 2026-04-05.

[20] MongoDB, Inc. Mongodb: The developer data platform, 2024.

[21] Amazon Web Services, Inc. Amazon DynamoDB: Managed NoSQL database service, 2026.

[22] Pramod J. Sadalage and Martin Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley Professional, Upper Saddle River, NJ, 2012.

[23] Google LLC. Cloud functions for firebase. `https://firebase.google.com/docs/functions`, 2026. Accessed: 2025-01-01.

[24] 0xlimon. Lack of several features in self-host. GitHub Discussion #12151, Supabase, January 2023. Accessed: 2026-04-12.

[25] Joe Nelson and Steve Chavez. PostgREST 14 documentation, 2024. Accessed: 2025-02-16.

[26] Supabase. Auto-generated REST API via PostgREST, 2024. Accessed: 2025-04-04.

[27] Clockwork Labs. Spacetimedb and bitcraft, 2023.

[28] Clockwork Labs. Spacetimedb, 2024. Accessed: 2026-04-12.

[29] Vicent Martí. SpacetimeDB: a short technical review. `https://strn.cat/w/articles/spacetime/`, February 2026. Accessed: 2026-03-22.

[30] Clockwork Labs. SpacetimeDB: Keynote 2 benchmarks. `https://github.com/clockworklabs/SpacetimeDB/tree/master/templates/keynote-2`, 2026.

[31] Gani Georgiev. PocketBase: Open source realtime backend in 1 file, 2026.

[32] Gani Ganev. Going to production — PocketBase documentation. Accessed: 2026-04-12.

[33] JSON-RPC Working Group. JSON-RPC 2.0 specification, January 2013. Originally published 2010-03-26. Accessed: 2026-04-14.

[34] Benedikt Spies and Markus Mock. An evaluation of webassembly in non-web environments. In *2021 XLVII Latin American Computing Conference (CLEI)*, pages 1–10. IEEE, 2021.

[35] Duncan Uszkay. How Shopify uses WebAssembly outside of the browser, December 2020. Accessed: 2026-04-14.

[36] Steve Manuel. Beyond the HTTP API: WebAssembly and the future of systems integration. Dylibso Blog, aug 2024. Accessed: 2026-02-16.

[37] Tetrate.io, Inc. wazero: The zero dependency WebAssembly runtime for Go developers, 2026. Version 1.11.0. Apache-2.0 License. Accessed: 2026-04-14.

[38] Benchmarks Game Team. Measured: Which programming language is fastest? (The Computer Language 25.03 Benchmarks Game), 2025. Accessed: 2026-04-14.

[39] Rob Pike. Go at Google: Language design in the service of software engineering, October 2012. Keynote talk at SPLASH 2012, Tucson, Arizona. Accessed: 2026-04-14.

[40] Muhammad Waseem, Teerath Das, Aakash Ahmad, Peng Liang, and Tommi Mikkonen. Issues and their causes in webassembly applications: an empirical study. In *Proceedings of the 28th international conference on evaluation and assessment in software engineering*, pages 170–180, 2024.

[41] Linus Wagner, Maximilian Mayer, Andrea Marino, Alireza Soldani Nezhad, Hugo Zwaan, and Ivano Malavolta. On the energy consumption and performance of webassembly binaries across programming languages and runtimes in iot. In *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, pages 72–82, 2023.

[42] WebAssembly Community Group. WebAssembly garbage collection (GC) proposal, 2024. Accessed: 2026-04-14.

[43] Alex Petrov. *Database Internals: A deep dive into how distributed data systems work*. O'Reilly Media, 2019.

[44] dgraph-io. Badger. `https://github.com/dgraph-io/badger`, 2017. Accessed: 2026-04-18.

[45] Ben Johnson and etcd-io contributors. bbolt: An embedded key/value database for Go, 2025. Version 1.4.3. MIT License. Accessed: 2026-04-14.

[46] Conventional Commits. Conventional commits 1.0.0, 2019. Accessed: 2026-04-21.

[47] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[48] MongoDB, Inc. Read isolation, consistency, and recency. `https://www.mongodb.com/docs/manual/core/read-isolation-consistency-recency/`. MongoDB Manual, Version 8.2. Accessed: 2026-04-27.

[49] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[50] Mike Wacker. Just say no to more end-to-end tests. `https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html`, April 2015. Google Testing Blog. Accessed: 2026-04-18.

[51] Kent C. Dodds. Write tests. not too many. mostly integration. `https://kentcdodds.com/blog/write-tests`, July 2019. Accessed: 2026-04-18.

[52] gin-gonic. Gin: Http web framework written in go. `https://github.com/gin-gonic/gin`, 2026. Accessed: 2026-04-20.

[53] Cindy Sridharan. *Distributed Systems Observability.* O'Reilly Media, 2018.

[54] Bytecode Alliance. RFC: Wasmtime debugging. `https://github.com/bytecodealliance/rfcs/blob/main/accepted/wasmtime-debugging.md`. Accepted RFC in the `bytecodealliance/rfcs` repository. Accessed: 2026-04-27.

Appendix A

Self-appraisal

A.1 Critical self-evaluation

I am ultimately happy with my process during the software development of this project. I started programming in October and was able to de-risk my project by validating the core concepts surrounding the execution environment early. As a result of this, I am proud of the software I have built. I think it is well engineered, functional, and utilises modern technology in a unique and interesting way.

Starting as early as the summer before semester one, I knew what exactly I wanted to build. However the problem I was trying to solve was sometimes vague and undefined. While I believe this project does take a unique perspective on solving the problem of web application backend development, it took a while for me to appropriately contextualise it verses alternatives in the industry. It is one thing to know what you want to build, and another to articulate precisely why it needs to exist and what gap it fills. At times I feel that I was overly focused on the technical implementation of my solution, losing track of the wider problem I was attempting to solve.

The choice to write large parts of the database component from scratch was, while very fun and interesting, ultimately not the focus of the project and resulted in a less performant and featured project.

If given the chance to do this project again, I would have formalised my literature review earlier, this would have given me a much better context of the area of the industry my project occupies. I would also consider using more off-the-shelf software components for the database, speeding up development time, freeing me to focus on the report earlier.

A.2 Personal reflection and lessons learned

I feel that during this project I have developed the skill of researching a lot and have engaged with the academic side of computer science more than previously. While researching I had the pleasure of reading some extremely influential and highly cited papers which helped me gain a greater appreciation the amount of research that was required to develop our modern computing systems.

With that said, I was humbled somewhat by the complexity encountered in the field of databases research. While I accepted that this project was not going to be easy, concepts such as concurrency control and query optimisation required significantly more research than I expected. I gained a greater appreciation for how my project only scratches the surface of this field of study as well as the intelligence and work required to build and maintain database

software I take for granted.

More practically, this project reinforced to me the importance of scoping. The decision to implement database components from scratch, while intellectually rewarding, made clear to me that this ambition must be balanced against the constraints of time and that the core goals of the project must be prioritised. That is perhaps the most transferable lesson I take from this experience.

In future projects, I would set aside a few more weeks prior to beginning implementation that would be dedicated to more in-depth academic research into the context of the problem I was attempting to solve. This would ensure the context to the problem was clearly defined prior to design decisions being made. I mistakenly treated the literature review as a sort of background task to be done between programming sprints rather than a core milestone to have completed early in development, and this lack of understanding of existing solutions resulted in some features being prioritised incorrectly.

A.3 Legal, social, ethical and professional issues

A.3.1 Legal issues

During the development of LEODIS, great effort was taken to ensure that any external libraries or dependencies were correctly licenced for use within this project. LEODIS itself is licenced under an MIT licence, a highly permissive open-source licence.

The question of open-source licencing in the field of web technology is particularly interesting at this point in time. MongoDB, used consistently as a point of comparison to LEODIS, switched from the open-source AGPL licence to the Server Side Public Licence in 2018. This was in an attempt to prevent hyperscalers like Amazon providing hosted MongoDB instances which was seen as unfairly taking the economic value of open-source developers.

This tension between open-source and commercial sustainability is an unresolved legal and ethical debate in the software industry. For a project like LEODIS, which is currently academic in nature, a permissive license such as MIT is appropriate.

A.3.2 Social issues

Since LEODIS can be used for storing data, there is some concern for how it could be misused for the storage of illegal or socially harmful content. However due to the experimental nature of this project, the likelihood of it being used for this is low.

While this report itself deals with a technical concepts, it was designed for consumption by a diverse audience, and great effort was taken to ensure it is accessible to people of all backgrounds.

A.3.3 Ethical issues

As LEODIS contains a database that could be used for storing sensitive data, there is a responsibility to ensure that vulnerabilities which could compromise the confidentiality of that

data are either patched or that users are made aware so they can mitigate the risk.

Responsible disclosure is an established practice in the software industry. Once a vulnerability is discovered, it should be reported privately to the maintainer before being made public, allowing time for a fix to be developed. As the author and maintainer of LEODIS, I would follow this principle and take security reports seriously.

Additionally, the decision to give LEODIS the MIT licence keeps the system transparent and observable, allowing other developers to view and validate the security of the system.

A.3.4 Professional issues

As a software project intended to be released openly, LEODIS carries professional responsibilities consistent with those outlined in the BCS Code of Conduct, which requires practitioners to act in the public interest and with due care and diligence. Specifically, the Code requires that members ”have due regard for public health, privacy, security and wellbeing of others”. The great efforts taken to ensure LEODIS is a secure and reliable system reflects a duty of care to users ensuring that buggy stored procedures cannot compromise the host system or user data.

Appendix B

External Material

This section contains lists of all external libraries, tools, and assets used in the development and evaluation of LEODIS. All other components were developed entirely by the author.

Library	**Purpose**	**Licence**
Wazero	WebAssembly runtime	Apache 2.0
Badger	Primary storage engine	Apache 2.0
Bbolt	Legacy storage engine	MIT
Gin	HTTP framework	MIT
UUID	UUID generator	BSD 3-Clause
Serde	Serialisation and deserialising Rust data structures	Apache 2.0

Table B.1: External libraries for the implementation of LEODIS

Tool	**Purpose**
k6	Load testing and benchmarking
PocketBase	Comparison BaaS implementation
MongoDB	Database for comparison three-layer implementation

Table B.2: External tools used for evaluation

Tool	**Purpose**
Pandas	Data manipulation and analysis
Matplotlib	Chart and figure generation
Seaborn	Statistical data visualisation
Geist Mono	Font used in figures and code listings

Table B.3: External tools used for data visualisation

Library	Purpose	Licence
@tanstack/react-router	Client-side routing	MIT
@tanstack/router-devtools	Routing development tools	MIT
lucide-react	Icon components	ISC
react	UI framework	MIT
react-dom	React DOM rendering	MIT
react-syntax-highlighter	Code and log syntax highlighting	MIT

Table B.4: External libraries used for the admin user interface

Appendix C

Example code

Figure C.1: Example of a stored procedure that returns the average of a field in the database

```
//go:build wasm

package main

import (
        sdk "leodis/function/sdk/go"
)

// Stored procedure entrypoint
//
//go:wasmexport run
func run() uint64 {
        // Find all cats (no filter)
        q := sdk.Query{
                Filter: sdk.Filter{},
                Sort:   sdk.Sort{},
                Limit:  -1,
        }

        docs, err := sdk.FindRaw("cats", q)

        if err != nil {
                sdk.Log(err.Error())
                return 0
        }

        if len(docs) == 0 {
                return sdk.PackResponse(sdk.Document{"average_age": 0.0})
        }

        totalAge := 0.0
        for _, doc := range docs {
                age, _ := doc["age"].(float64)
                totalAge += age
        }
        averageAge := totalAge / float64(len(docs))

        sdk.Log("Query␣complete")

        // Return data back upstream to host, to be returned to the client
        return sdk.PackResponse(sdk.Document{"average_age": averageAge})
}
```

Figure C.2: Example of a stored procedure that searches for a document by a field

```
//go:build wasm

package main

import (
        sdk "leodis/function/sdk/go"
)

//go:wasmexport run
func run() uint64 {
        searchName := sdk.GetInput()
        sdk.Log(searchName)

        // Query using the extracted name
        q := sdk.Query{
                Filter: sdk.Eq("name", searchName),
                Sort:   sdk.Sort{},
                Limit:  -1,
        }
        packed, err := sdk.FindRawPacked("cats", q)
        if err != nil {
                sdk.Log(err.Error())
                return 0
        }

        return packed
}
```