DELFT UNIVERSITY OF TECHNOLOGY

MASTERS THESIS

Obol: From Imperative Code to Distributed Workflows

Author:
Stavros Alexandros Dimakos

Supervisor:
Dr. Asterios Katsifodimos

Daily Co-Supervisor:
Dr. Jeff Smits

A thesis submitted in fulfillment of the requirements
for the degree of MSc Data Science and Artificial Intelligence Technology

in the

Web Information Systems Group
Software Technology

Thesis committee:

Chair:	Dr. Jesper Cockx
Committee member:	Dr. Annibale Panichella
Supervisor:	Dr. Asterios Katsifodimos
Daily co-supervisor:	Dr. Jeff Smits

June 30, 2026

"If you think you have a limit, as soon as you touch this limit, something happens, then you suddenly can go a little bit further. With your mind power, your determination, your instinct and your experience as well, you can fly very high."

Ayrton Senna

DELFT UNIVERSITY OF TECHNOLOGY

Abstract

Electrical Engineering, Mathematics and Computer Science
Software Technology

MSc Data Science and Artificial Intelligence Technology

Obol: From Imperative Code to Distributed Workflows

by Stavros Alexandros Dimakos

Serverless computing has transformed how distributed applications are built, enabling developers to deploy event-driven, scalable services without managing infrastructure. Early work on extending this paradigm to stateful applications through Stateful Function-as-a-Service (SFaaS) has shown that co-locating state with compute, effectively eliminating the "shipping data to code" problem, yields significant gains in throughput and latency. Systems such as Styx demonstrate that SFaaS can deliver serializable, exactly-once transactional guarantees across arbitrary function call-graphs at high performance. However, these performance benefits come at a cost: developers must decompose naturally sequential workflows into chains of asynchronous callbacks, manually managing continuations, context serialization, and distributed control flow, pulling attention away from business logic and toward distributed coordination.

In this thesis, we present Obol, a compiler-driven approach that raises the level of abstraction for SFaaS programming. Obol allows developers to express distributed stateful workflows as ordinary sequential, object-oriented code, and automatically compiles it to the asynchronous message-passing form required by the underlying runtime. We show that a multi-stage compiler pipeline can translate standard object-oriented constructs into correct distributed dataflow programs without sacrificing the transactional guarantees of the target runtime. We evaluate Obol on the YCSB and TPC-C benchmarks and demonstrate that the compiled code tracks hand-written operator latency to within a few percent up to saturation and saturates only modestly earlier. The residual gap is not compilation overhead, which a control workload isolates at essentially zero, but the cost of Obol's structured reply routing relative to hand-tuned callback wiring, a cost that the available concurrency constructs and optimizations significantly decrease.

Acknowledgements

With the completion of this thesis, I close a very important chapter of my life, concluding the past five years of my bachelor's and master's studies at TU Delft. I am deeply grateful for the opportunities and experiences I have gained during this time. I had the chance to meet and learn from people from many different backgrounds and to realise first-hand that we are not so different after all. I was able to grow both personally and technically, and I hope that I have been able to contribute to others in return, just as many have contributed to my own development. I am grateful to all who have been part of this journey, both peers and professors alike.

I want to give a special thanks to the people who helped me finalize my time here. I want to thank my professor, Asterios Katsifodimos, for being so supportive, curious, and excited about my work, but more importantly for making me feel comfortable and engaged with it from the very first day (and for the pizzas, of course). I want to thank my supervisor, Jeff Smits, not only for helping with the implementation by writing a package I could use for part of the analysis, but also for always looking out for my progress, making sure everything was going smoothly and that I kept moving forward. His guidance was truly appreciated. Finally, I want to thank Kyriakos Psarakis, first because without his implementation of Styx this thesis would not have been possible in the first place, and also for taking time out of his busy schedule to support, provide feedback and help with any issues I ran into. Beyond the direct help, all three created a genuinely welcoming environment. They were approachable and easy to talk to, and never made me feel like I shouldn't ask for help. I feel very fortunate to have worked with these people during my thesis.

I also want to thank my family for always supporting me unconditionally, for worrying even when they did not need to, and for always being excited about my progress and achievements. Finally, I want to thank my friends, it was truly an amazing experience to see how we all shaped our separate paths and how we learned from each other.

Contents

Chapter 1

Introduction

Consider one of the simplest operations an e-commerce store can perform: a user buys an item. In code, it reads exactly as a programmer would naturally write it, a balance check, a stock update, and a few assignments:

```python
def buy_item(self, amount: int, item: Item) -> bool:
    total_price = amount * item.get_price()
    if self.balance < total_price:
        raise NotEnoughBalance("Not enough balance.")

    item.update_stock(-amount)
    self.balance -= total_price
    self.myitems.append(item)
    return True
```

LISTING 1: Python implementation of a buy_item transaction

Run this on a single machine, and it works. But a single machine only takes you so far: as load grows, the `User` and `Item` entities have to be spread across separate workers, and the moment they are, the code above stops working. The two now communicate over the network instead of shared memory, so each cross-entity call becomes a message, the local stack frame is destroyed at every suspension point, and the method must be decomposed by hand into a chain of callbacks, each carrying its own serialized context dictionary, return-address record, and explicit state-persistence call [9]. This is the *stack-ripping problem* [14], resurrected at a new layer of the stack.

Existing approaches to this gap create certain trade-offs. Workflow orchestrators such as AWS Step Functions [3] and Apache Airflow [1] push coordination into a separate declarative artifact that lives outside the application, breaking the locality that object-oriented code provides. Durable execution systems such as Azure Durable Functions [5] and Temporal [44] keep coordination in the application's language but achieve durability through deterministic replay: the runtime journals an execution's effects and reconstructs failed runs by re-executing the code against that history. For replay to be sound the code must be deterministic, so it cannot read the clock, generate random numbers, use native concurrency, or perform I/O outside SDK-provided wrappers. In both cases, the developer pays for durability either by leaving the language or by writing in a restricted dialect of it.

Recent stateful Function-as-a-Service (SFaaS) runtimes such as Styx [32] represent a closer approximation. Styx co-locates state with compute, provides serializable transactions across arbitrary function call-graphs, and achieves exactly-once semantics through asynchronous barrier snapshotting [7]. Durability comes from the runtime rather than from replay, and the language remains unrestricted. However, the developer-facing API remains low-level: cross-entity interactions are still

expressed as explicit asynchronous message dispatches, and the eight-line method above (Listing 1) must still be authored by hand as separate event-driven callbacks.

The most direct attempt to raise this level of abstraction is Stateful Entities [51], which first showed that imperative object-oriented Python can be compiled into a stateful dataflow by splitting each method at its remote-call boundaries. It establishes the feasibility of the source-to-dataflow approach we build on, but coordinates cross-entity calls through a centralized execution graph and inherits only the message-level guarantees of the target dataflow engine, leaving serializable cross-entity transactions, recursion, and concurrent dispatch out of reach.

We present **Obol**, a compiler that closes the remaining gap. Obol takes type-annotated sequential Python and produces an equivalent program of asynchronous Styx operator functions connected by explicit message-passing. The transformation threads live variables through a backward dataflow analysis, reifies loops as tail-recursive step functions, reconstructs the distributed call stack as a serializable record, and supports fan-out of independent calls into persistent synchronization barriers, all without weakening the underlying runtime's transactional guarantees. The developer writes the eight-line transaction; the compiler produces the distributed counterpart.

The contributions of this work are as follows:

- **A high-level programming model for distributed stateful applications.** We define a class-based, imperative programming model for distributed stateful workflows, realized on an SFaaS runtime. Entities are expressed as annotated Python classes, with state managed implicitly and cross-entity interactions written as standard method calls. The model supports common Python control flow constructs, including loops, nested conditionals, and recursion.
- **A CPS-based compiler for imperative Python control flow.** We present a multi-stage pipeline that translates high-level entity methods into Styx-compatible operator chains, handling arbitrarily nested control flow and asynchronous fan-out, with a live-variable analysis used to minimize the context serialized across asynchronous boundaries.
- **An evaluation on standard transactional workloads.** We evaluate Obol on YCSB [10] and TPC-C [46] against hand-written Styx operator code, showing that the compiled code stays close to hand-written Styx, saturating only marginally earlier, when the available concurrency constructs are used.

Obol is evidence for a broader claim: the stack-ripping problem in distributed systems is not a fundamental constraint of the programming model, but a consequence of working at the wrong level of abstraction.

The full implementation of Obol is available in the Styx repository at: `https://github.com/delftdata/styx/tree/main/obol`.

Chapter 2

Background

2.1 Function-as-a-Service

FaaS represents the departure from traditional on-premise infrastructure toward serverless computing. By abstracting the underlying hardware and middleware, FaaS allows developers to deploy modular code without the operational burden of server management. In this ecosystem, the cloud provider dynamically manages the allocation of resources, enabling developers to focus exclusively on application logic rather than backend maintenance [23].

The primary advantage of FaaS is its improvement of operational efficiency and cost management. Because functions are inherently event-driven, there is no cost associated with idle server time between executions: resources are provisioned only when triggered by a specific event, such as an API call or a database update, and are immediately decommissioned once the task is complete. This pay-per-execution model eliminates the financial waste of maintaining over-provisioned hardware. FaaS also provides inherent scalability, as the cloud provider automatically manages the underlying compute power to meet fluctuating demand, allowing developers to focus entirely on code quality rather than capacity planning. Popular FaaS solutions include AWS Lambda, Azure Functions, and Google Cloud Functions [13].

2.2 Stack Ripping

In traditional synchronous programming, the compiler automatically manages the execution state of a function using the call stack. Each stack frame tracks the instruction pointer, local variables, and return addresses, providing a linear and predictable execution trace. However, this model becomes untenable in event-driven architectures. Because asynchronous functions must return control to the event loop while waiting for external events, the local stack frame is destroyed, forcing the programmer to manually capture and preserve state in the heap. This process, known as *stack ripping*, requires fragmenting single logical tasks into multiple continuation objects or callbacks. Early research identified this as a critical issue because it obscures the natural control flow, making it significantly more difficult for developers to reason about, debug, and maintain complex applications [14, 47, 26].

To address these challenges, research projects such as Tame [25] demonstrated that a compiler could automate state management by leveraging coroutines. Unlike standard functions that adhere to a strict last-in-first-out execution order, coroutines act as generalized control structures that allow execution to be suspended and later resumed at specific yield points. By transforming seemingly sequential code into a stackless coroutine state machine, the compiler can automatically lift local variables from the ephemeral stack into a persistent heap-allocated object. This evolution

eventually culminated in the widespread adoption of the `async/await` paradigm, which hides the underlying stack ripping from the developer by maintaining the illusion of a continuous call stack while execution is multiplexed by an event loop.

However, the rise of serverless and stateful FaaS runtimes has brought the problem of stack ripping full circle, though at a different layer of the stack. In these serverless environments, a single logical workflow is executed across multiple physical machines, meaning execution state cannot simply reside in the local memory of a single process. Because a traditional call stack cannot exist between different servers, state must be explicitly captured, serialized into a portable format, and moved across the network to a new execution context. Recent research, such as the Beldi [49] system, highlights that this necessitates distributed fault tolerance where every operation must be atomically logged alongside its current step number to ensure exactly-once semantics. In this context, the developer is once again forced to adopt event-driven abstractions to manage state across distributed boundaries, as the lack of a unified distributed coroutine mechanism recreates the original stack-ripping dilemma on a macro scale [19].

2.3 Stateful Serverless and Durable Execution

To address certain limitations of stateless FaaS, the Stateful Function-as-a-Service (SFaaS) paradigm co-locates state with compute. Rather than externalizing persistent state to a remote database, an SFaaS runtime manages state directly alongside the functions that operate on it, routing each invocation to the worker that owns the relevant data. Existing systems fall into three broad design families.

Log-based SFaaS Beldi [49], Boki [22], and Halfmoon [34] provide exactly-once semantics by journaling every invocation and state operation on a durable log. Beldi runs on unmodified commercial FaaS platforms; Boki is a purpose-built runtime that exposes a shared-log API to functions; and Halfmoon shows that, rather than symmetrically logging both reads and writes, it suffices to log only one side asymmetrically while still preserving exactly-once semantics. In all three, developers write business logic against a wrapper library that issues journaled read, write, invoke, and transaction calls.

Durable execution Another family lets developers write *orchestrator* functions in a general-purpose language. The runtime journals the orchestrator's decisions to a persistent event history and, on failure or rescheduling, replays the code from the start, skipping completed calls by returning their recorded results from the history. Azure Durable Functions [5] and Temporal [44] are the canonical examples. Replay correctness requires the orchestrator to be deterministic: no direct clock reads, no random number generation, no native concurrency, and no side-effectful I/O outside SDK-provided wrappers.

Dataflow-based SFaaS A third family co-locates state with computation on top of a streaming-dataflow runtime. Apache Flink Stateful Functions [45] exposes addressable stateful functions over Flink and inherits exactly-once processing from asynchronous barrier snapshotting [7], an adaptation of the Chandy–Lamport global-snapshot protocol [8] for acyclic streaming dataflow graphs. Cloudburst [40] also belongs to this family in spirit, co-locating Python functions with caches over Anna, a key-value store, but its compute-tier fault tolerance is whole-DAG re-execution

rather than barrier snapshotting. Styx [32] belongs to this family and is the runtime we target: it combines asynchronous barrier snapshotting with a deterministic transactional protocol, giving arbitrary function call-graphs serializable, exactly-once semantics without two-phase commit. We target Styx because it offers state-of-the-art SFaaS performance while providing exactly the transactional guarantees Obol requires.

2.3.1 The Low-Level Operator API

Styx provides a Python API based on two abstractions: the `Operator` and the `StatefulFunction`. An `Operator` represents a partitioned stateful entity, similar to a database table or actor class. Its key space is distributed across workers. Functions are registered using `@operator.register` and act as entry points.

Each function receives a `ctx` object of type `StatefulFunction`, which provides access to state and communication:

- `ctx.get()` returns the value at the current key, or `None`.
- `ctx.put(value)` updates the value at the current key.
- `ctx.key` returns the current key.
- `ctx.batch_insert(dict)` inserts multiple key-value pairs.
- `ctx.call_remote_async(operator_name, function_name, key, params)` invokes a function on another operator.

The `call_remote_async` primitive enables cross-entity transactions. All calls are tracked as part of the same transaction, which completes only after all invoked functions finish. Since it does not return a value, operations that require results must be split into two functions. The remote function returns results by calling a second function on the original operator. This pattern reflects the stack-ripping problem (Section 2.2), where sequential logic is split into callbacks. The high-level model presented next (Chapter 3) automates this process.

2.4 Programming Languages

Continuation-Passing Style. In standard direct-style programming, a function computes a value and returns it to its caller implicitly via the call stack. *Continuation-Passing Style* (CPS) makes this control flow explicit: instead of returning a value, every function receives an additional argument, the *continuation*, representing the remainder of the computation, and invokes it with the result upon completion. First used as a compiler intermediate representation by Steele [41] and later systematized by Appel [2] and revisited by Kennedy [24], CPS has the useful property of flattening nested expressions into a strict sequence of named intermediate steps, making every control transfer syntactically visible. Plotkin's correctness results [30], namely Indifference, Simulation, and Translation, establish that the CPS transform faithfully encodes source-level evaluation and that CPS-transformed terms are themselves insensitive to evaluation order. In our distributed setting, this evaluation-order insensitivity enables the scheduling freedom the runtime requires. This makes CPS the natural tool for compiling `async/await`: the compiler automates the transformation that a developer would otherwise perform manually when splitting a function

at each suspension point. The technique underlies `async/await` in F# [43], asynchronous C# [4], JavaScript, Python (PEP 492) [36], and C++20 coroutines; it is also the standard tool for stackless web continuations [29]. Obol applies the same mechanism, where its target is not a thread-local event loop but a distributed runtime where continuations travel as network messages.

A-Normal Form. A direct-style counterpart of CPS is *A-Normal Form* (ANF), introduced by Flanagan, Sabry, Duba, and Felleisen [15]. In ANF, every non-trivial subexpression is hoisted into a `let`-bound temporary so that evaluation order is syntactically explicit: `f(g(x), h(y))` becomes `let t1 = g(x) in let t2 = h(y) in f(t1, t2)`. Flanagan et al. showed that ANF and CPS are closely related via semantics-preserving translations up to administrative reductions, and that many CPS-level optimizations can be expressed as corresponding ANF transformations.

Defunctionalization. A continuation in CPS is naturally represented as a closure, a function paired with the local variables it captures from its lexical environment. While closures are convenient within a single process, they are typically non-portable, since standard implementations rely on in-memory function pointers and heap addresses that cannot be serialized across a network or written to persistent storage. *Defunctionalization*, introduced by Reynolds [35] and later formalized by Danvy and Nielsen [12] and extended to polymorphic settings by Pottier and Gauthier [31], is a whole-program transformation that eliminates higher-order functions by replacing each distinct closure with a unique first-order data tag and lifting its captured variables into an explicit record. A single global *apply* function then dispatches on the tag to recover the original behaviour. The result is a program that passes only plain, inspectable data structures in place of opaque function values, which is a prerequisite for continuations that must survive serialization to persistent storage or transmission across a network.

Chapter 3

The Obol Programming Model

Object-oriented programming endures as the dominant paradigm for application code because it mirrors how domain experts describe their problems: stateful entities with identities, encapsulated behaviour, and interactions written as ordinary method calls. Programmers reach for it precisely because the code reads as the business logic it encodes. This alignment breaks down once those entities are distributed, and the developer is forced back into callbacks, manual continuations, and explicit message dispatch. Obol's design goal is to restore the alignment: a developer should write a distributed stateful workflow as ordinary sequential, object-oriented Python, and the compiler should produce a distributed program with the same semantics. The surface remains familiar (classes, methods, fields, calls) while the compiler handles the routing, state persistence, and continuation synthesis that would otherwise need to be written by hand. The abstraction must do this without weakening the transactional guarantees of the underlying runtime: every property a developer relies on when writing against the low-level Styx API must continue to hold when writing against Obol.

This section describes the resulting programming model. We first introduce what the developer writes (Section 3.1), then the coordination work the model handles implicitly (Section 3.2), followed by the control flow and concurrency constructs that span both (Section 3.3). A short discussion of the semantics Obol preserves closes the section.

3.1 Writing Obol Programs

We illustrate the model through a small e-commerce application used as a running example throughout the rest of the paper. The application has two central entities: an `Item` representing a product with a name, price, and stock level, and a `User` representing a customer with a balance and a list of purchased items. The main transaction is `buy_item`, in which a `User` retrieves an `Item`'s price, checks affordability, decrements stock, and records the purchase.

In Obol, this reads as:

```python
@entity
class User:
    def __init__(self, name: str, balance: int):
        self.name = name
        self.balance = balance
        self.myitems: list[Item] = []

    def __key__(self) -> str:
        return self.name

    def buy_item(self, amount: int, item: Item) -> bool:
        total_price = amount * item.get_price()
        if self.balance < total_price:
            raise NotEnoughBalance("Not enough balance.")
        item.update_stock(-amount)
        self.balance -= total_price
        self.myitems.append(item)
        return True
```

LISTING 2: The `User` entity and its `buy_item` method as written in Obol.

Four constructs make up the surface.

Entities. A distributed stateful object is a Python class annotated with `@entity`. Its instance attributes form the entity's persistent state, declared in `__init__` just as in ordinary Python. There is no separate schema language and no state-store API to learn.

Keys. Every entity defines a `__key__` method returning its routing key. The runtime uses this key to partition instances across workers and route messages. Composite keys are expressed as tuples: a `Review` entity jointly identified by a user and an item returns `(self.user_name, self.item_name)` from `__key__`. The type checker verifies that any key passed in matches the tuple's component types.

Methods. Methods are plain Python functions over `self` and parameters. They are written to execute sequentially from start to finish. The developer never writes `async def`, never awaits a remote call, and never serializes a context dictionary by hand.

Calls. A call from one entity to another, such as `item.get_price()` in `buy_item`, is written identically to a local method call. The developer does not name a target operator, construct a message, or register a callback. From their viewpoint, the call returns a value and execution proceeds with that value in scope.

When a developer needs to obtain a reference to a remote entity by key rather than receiving one as a parameter, the model provides the `get_entity_by_key` helper, which serves as a compile-time hint carrying the target entity type and key. The compiler replaces each occurrence with the appropriate runtime key reference.

3.2 The Programming Surface

The four surface constructs are deliberately sparse. Everything else, the coordination work that makes a distributed program correct, is handled implicitly.

State persistence. The developer never calls a `save()` or `commit()` method. Modifications to an entity's instance attributes are persisted by the runtime before each remote dispatch, so any worker that subsequently observes the entity sees its latest state.

Routing. Every cross-entity call is routed to the worker that owns the callee's key, transparently to the caller. Whether the callee resides on the same worker or a remote one has no syntactic effect.

Continuation management. The developer writes no context dictionaries, names no callback functions, and manages no distributed call stack. The compiler partitions each method at its remote call sites and synthesizes the chain of step functions that carries live variables across each asynchronous boundary. The mechanism is described in Chapter 4.

Type-driven dispatch. No annotation marks a call as remote. The compiler infers it from the type of the receiver: a call on a value whose type resolves to an `@entity` class compiles to an asynchronous dispatch; everything else compiles to an ordinary local call. The programmer writes one kind of call, and the type system decides which it is.

3.3 Control Flow and Concurrency

To express realistic workflows, the model must go beyond straight-line sequences of calls: it supports branching, loops, and recursion, as well as independent calls that need not block on one another.

Standard control flow. Entity methods may use Python's ordinary control flow: `if`/`else`, `while` and `for` loops, recursion, and `break`/`continue`. Remote calls may appear at any point within these constructs, including inside loop bodies, across branches of a conditional, or along a recursive call chain. The developer writes the loop or the recursion as they would in a single-process program; the compiler is responsible for preserving its semantics across the asynchronous boundaries introduced by each remote call. Chapter 4 describes how this is done, and Chapter 6 lists the Python constructs that fall outside the supported subset.

Concurrent dispatch. Strictly sequential composition forfeits a key advantage of a distributed runtime: independent remote calls can travel in parallel. Obol exposes two constructs for this. The first is a fire-and-forget dispatch, which issues a remote call without registering a continuation and lets the caller proceed immediately; the call's return value is suppressed and cannot be observed by the caller. The second is a fan-out/fan-in primitive that issues a group of independent remote calls concurrently and binds their results once all have completed, with total latency bounded by the slowest of the round-trips rather than their sum. The semantics mirror Python's `asyncio.gather`, but the underlying synchronization barrier is persistent and survives worker failure. Chapter 4 describes how both constructs are compiled.

Asynchrony at the call site, not the definition. This is a deliberate design choice. Every cross-entity call in Obol is asynchronous at the runtime level, but the programming model expresses this distinction at the call site rather than at the method definition. A bare call compiles to an asynchronous dispatch whose continuation awaits the reply, recovering await-style sequential composition without requiring every method to be annotated `async`. A `send_async` call compiles to a dispatch with no continuation. The result is a single uniform method signature, with the concurrency mode decided where the call is made.

Preservation of runtime semantics. The abstraction imposes no additional distributed coordination cost. Each cross-entity call in the source compiles to exactly one asynchronous dispatch in the output; no additional round-trips, state reads, or coordination messages are introduced by the translation. State partitioning remains entity-local, and the generated step functions execute directly within the Styx dataflow engine, inheriting its serializable, exactly-once transactional guarantees without modification. The abstraction removes the burden of expressing these properties manually; it does not weaken them.

Chapter 4

Architecture

The compiler is organized as a multi-stage transformation pipeline, shown in Figure 4.1. It takes type-annotated Python source code and produces an equivalent program made up of stateful operator functions connected through explicit asynchronous message passing.

The core of the pipeline is the *function splitting* pass, which partitions each method at its remote-call boundaries into a chain of operator functions. This pass has two prerequisites: a flat statement list with at most one remote call per statement, and statically resolved receiver types to determine each continuation's target operator. The stages that precede it exist to establish both, so we present the pipeline in execution order. We first describe the syntactic preparation passes that flatten each method into a linear statement list (Section 4.1). We then cover the type resolution pass that identifies which call sites target remote entities (Section 4.2) and the reply-to stack that routes return values across asynchronous message boundaries (Section 4.3). With both prerequisites in place, we present function splitting itself (Section 4.4) and its extension to loops, branches (Section 4.5) and recursion (Section 4.6). Finally, we describe the constructs Obol provides for concurrent dispatch (Section 4.7), which let independent remote calls overlap rather than execute strictly in sequence.

The transformation is based on continuation-passing style (CPS) and defunctionalization. Instead of returning values through the call stack, each function passes its result forward to explicitly named step functions. Any local variables that would normally be captured in a closure are stored in a serializable context record. The generated `step` functions act as defunctionalized continuations.

All stages operate on a shared intermediate representation based on the Python concrete syntax tree, produced by LibCST [21]. This representation is processed through a series of visitors and transformers. The pipeline assumes that the input program is type correct. Because Python's type system is optional, the compiler requires explicit type annotations wherever stateful objects are used, and it rejects programs that cannot be fully resolved through static analysis.

4.1 Flattening and Comprehension Expansion

The function splitter can only locate split points (remote call boundaries) if every remote call appears as a standalone, top-level assignment statement within a linear block. Three preparatory passes establish this invariant before function splitting takes place.

The first is a *comprehension expansion* pass, which rewrites list, set, and dictionary comprehensions that contain entity method calls into equivalent explicit loops.

Generator expressions that contain entity method calls are instead rejected at compile time, since eagerly materializing a lazy generator would change its observable behaviour and break semantic equivalence.

As an example, consider the `inventory_value` method, which sums the prices of a user's items using a list comprehension:

```
def inventory_value(self) -> int:
    total_value = sum([item.get_price() for item in self.myitems])
    return total_value
```

Because `item.get_price()` is a remote call, the comprehension cannot survive into function splitting. The pass rewrites it into an explicit `for` loop that accumulates results into a temporary variable, which is then passed to `sum`:

```
def inventory_value(self) -> int:
    _comp_result_1 = []
    for item in self.myitems:
        _comp_result_1.append(item.get_price())
    total_value = sum(_comp_result_1)
    return total_value
```

The pass processes comprehensions outermost-first, iterating until none remain in the block. This handles nested cases such as `[f(x) for x in [g(y) for y in items]]`. Set and dictionary comprehensions are treated analogously, with the accumulator initialised as `set()` or `{}` respectively. The correctness of this expansion follows directly from Python's definition of comprehensions as syntactic sugar for explicit loops; the pass is essentially a desugaring step.

The generated for loop deliberately leaks its iteration variable into the enclosing scope, unlike a Python 3 comprehension, so that the splitter's live-variable capture can see and restore it across asynchronous steps.

We describe the linearization pass before the short-circuit guarding pass, since the latter exists to support the former. The *linearization* pass handles remote calls that appear as subexpressions within larger expressions, hoisting each one into a temporary assignment so that every remote call appears as a standalone statement. This can be considered a partial conversion to A-Normal Form.

```
total_price = amount * item.get_price()
```

is rewritten into two statements:

```
attr_1 = item.get_price()
total_price = amount * attr_1
```

This applies recursively to chained and nested calls. Beyond simple expression contexts, the linearizer must also handle remote calls that appear inside loop conditions. A loop-unaware extraction would hoist the remote call into an assignment before the loop header, evaluating it only once rather than on each iteration. To preserve the correct semantics, the linearizer instead rewrites the condition into the loop body, transforming the loop into an unconditional `while True` with an explicit `break`. For example:

```
def drain_stock(self, item: Item) -> int:
    total = 0
    while 0 < (item.get_stock() - 1):
        item.update_stock(-1)
        total += 1
    return total
```

is rewritten by hoisting the condition into the loop body:

```
def drain_stock(self, item: Item) -> int:
    total = 0
    while True:
        attr_2 = item.get_stock()
        if not (0 < attr_2 - 1):
            break
        item.update_stock(-1)
        total += 1
    return total
```

The remote call now appears as an assignment at the top of the loop body, so it is linearized on every iteration rather than evaluated once before the loop.

The linearization pass introduces a subtle semantic issue. Python's `and` and `or` operators short-circuit: the right-hand operand is only evaluated if the left-hand operand does not determine the result. A naive linearizer would hoist a remote call out of such a conditional context, causing it to execute unconditionally and potentially raising an error that the short-circuit was intended to prevent. Consider:

```
def is_in_stock(self, item: Optional[Item]) -> bool:
    return item is not None and item.get_stock() > 0
```

This case is handled by the second pass, *short-circuit guarding*, which preserves short-circuit semantics by introducing a temporary variable for the guarding condition and placing the remote call inside a conditional block:

```
def is_in_stock(self, item: Optional[Item]) -> bool:
    _sc_1 = item is not None
    if _sc_1:
        attr_1 = item.get_stock()
        _sc_1 = attr_1 > 0
    return _sc_1
```

The remote call is now only reached when the guarding condition holds. The pass handles both `and` and `or` chains, applies recursively to nested boolean expressions, and runs before the linearizer so that the resulting `if`-guarded calls are themselves handled correctly by the subsequent pass.

4.2 Type Resolution

Type resolution serves one essential purpose: identifying which call sites target *stateful remote entities* without requiring any explicit annotation or compile hint from the developer beyond the type signatures. In the input programming model, classes decorated with `@entity` represent distributed stateful objects managed by the runtime. Consider the `buy_item` method (Listing 2): the call `item.get_price()` is syntactically indistinguishable from any ordinary method call, but because `item` is typed as `Item` (an entity class) the compiler must transform it into an asynchronous remote invocation. The compiler uses the `mypy` metadata to resolve the type of the receiver at each call site and determine whether it belongs to a known entity class. Call sites that cannot be resolved due to missing annotations are rejected at this stage, enforcing the invariant that the full message-passing structure of the output program is determined statically.

4.3 The Reply-To Stack

Before describing function splitting, we look at how return values are routed back through a chain of asynchronous calls. In synchronous execution this is handled implicitly by the call stack: when function A calls B which calls C, the stack records where each callee must return to and the runtime unwinds it automatically. In a distributed setting no such shared stack exists, so the compiler must construct an explicit substitute that plays the same role across asynchronous message boundaries.

Every compiled function receives a `reply_to` parameter, which is a stack of *distributed return address records*. Each record is a dictionary containing four fields: `op_name` and `fun` identifying the operator and function to invoke when the current computation completes, `id` carrying the key of the entity instance to invoke it on, and `context` holding a unique id that keys into the callee's persistent function context store, where the saved local variables for that frame are held. This is defunctionalization in the sense of [35]: each continuation that would naturally be represented as a closure is replaced by a serializable first-order data record, and the generated step functions collectively act as the global apply function that dispatches on it.

Two generated helper functions manage this stack. `push_continuation` is called before a remote dispatch: it serializes the current local variables into the operator's persistent context store under a new id, appends the return address record to `reply_to`, and returns the updated stack. `send_reply` is called at the end of a function to return a value: it pops the top record from `reply_to` and fires an asynchronous remote call back to the address encoded in that record, passing the remaining stack along so the reply can propagate further up the call chain. If `reply_to` is empty the function is a top-level entry point and simply returns the result directly.

Together, the two helpers let a chain of compiled step functions maintain a distributed continuation stack with no centralized coordinator.

4.4 Function Splitting

With the preparatory passes in place, we reach the core of the pipeline: the function splitting pass. It partitions a method body at its remote-call boundaries into a chain of step functions, each executing synchronously up to a single remote call before suspending. At every boundary, the pass captures the variables live at that point and makes them available to the continuation that resumes once the remote call returns.

The Splitting Algorithm

For acyclic code — methods whose control flow graph contains no loops or branches — the splitting algorithm is straightforward. The pass scans the linearized statement list sequentially. When it encounters a method call on an instance whose type resolves to a known entity class, it recognizes this as a remote call and performs a split: the statements before the remote call remain in the current function, the remote call is replaced by an asynchronous invocation preceded by a `push_continuation` call, and all subsequent statements are moved into a new step function. Since no statement is ever revisited, the function can simply be cut at each remote call site, and a function with n remote calls produces exactly $n + 1$ functions in total. Loops and conditional branches require additional treatment and are discussed in Section 4.5.

Context Capturing

Because the runtime executes each step function as an independent invocation, possibly on a different worker, local variables do not persist across steps. The compiler therefore runs a backward may-analysis over each entity method's control-flow graph (CFG) to determine, at every split point, the minimal set of variables that must be preserved across the asynchronous boundary. A variable is live if some future path may read it before overwriting it: reading a name adds it to the live set, a definition removes it, and the set at a join is the union of the incoming sets. Names are resolved to local scope, so imports, attribute accesses, and globally bound identifiers are excluded. Each statement then records the variables live on entry and exit, and at every remote call the splitter intersects the live-out (live on exit) set with the variables actually defined so far along the current path, since a variable may be live across both branches of a conditional yet assigned in only one, to decide what to capture. For loops it also distinguishes the live set at the loop header, used when a continuation re-enters the loop, from the set at the loop exit, used when execution falls through. The analysis itself is provided by the `LibCST-dfa` package [38], a LibCST-based data-flow analysis framework that implements the monotone-framework approach of FlowSpec [39].

At each split point, the live variables are packaged into a serializable dictionary and stored in the operator's persistent function context under a unique identifier, and the corresponding return address is appended to the reply stack. The continuation step function receives this identifier as a parameter and restores the variables at the start of its body before proceeding. Once restored, the entry is removed from the context store as part of the same transactional step, so a completed frame leaves no residue and the store does not accumulate stale context across a transaction's lifetime. As a local optimization, when a continuation is invoked on the same operator without crossing a network boundary, the context dictionary is passed directly rather than written to and read back from the persistent store.

Example: `buy_item`

With all these steps in place, we can trace the full compilation of `buy_item`. The source method (Listing 3) issues two remote calls in sequence: `item.get_price()` to retrieve the unit price and `item.update_stock()` to decrement inventory. After the linearization pass, the expression `amount * item.get_price()` is split into a separate assignment, so the function body presents two top-level remote call statements. The splitting pass therefore produces three step functions (Listing 4).

```
def buy_item(self, amount: int, item: Item) -> bool:
    total_price = amount * item.get_price()

    if self.balance < total_price:
        raise NotEnoughBalance("Not enough balance.")
    item.update_stock(-amount)

    self.balance -= total_price

    self.myitems.append(item)
    return True
```

LISTING 3: Original `buy_item`

```python
@user_operator.register
async def buy_item(ctx: StatefulFunction, amount: int, item: str, reply_to: list =
↪  None) -> bool:
    reply_to = push_continuation(
        ctx, reply_to, 'user', 'buy_item_step_2', ctx.key,
        {'amount': amount, 'item': item}
    )
    ctx.call_remote_async(
        operator_name='item', function_name='get_price',
        key=item, params=(reply_to,)
    )

@user_operator.register
async def buy_item_step_2(ctx: StatefulFunction, func_context, attr_1=None, reply_to:
↪  list = None):
    __state__ = ctx.get() or {}
    params = resolve_context(ctx, func_context)
    (amount, item) = (params.get('amount'), params.get('item'))

    total_price = amount * attr_1
    if __state__['balance'] < total_price:
        raise NotEnoughBalance("Not enough balance.")

    # ... push_continuation to serialize live context ...
    ctx.put(__state__) # update entity state
    # ... dispatch to update_stock ...

@user_operator.register
async def buy_item_step_3(ctx: StatefulFunction, func_context,
↪  placeholder_return=None, reply_to: list = None):
    # ... restore context ...

    __state__['balance'] -= total_price
    __state__['myitems'].append(item)
    ctx.put(__state__)
    return send_reply(ctx, reply_to, True)
```

LISTING 4: Compiled output of `buy_item`

Figure 4.2 traces the runtime execution. `buy_item` pushes a return address record for `buy_item_step_2` onto `reply_to`, packages `amount` and `item` into the context payload, and fires the remote `get_price` call. The item operator responds to `buy_item_step_2` with `attr_1` bound to the returned price and `func_context` holding the unique identifier under which `amount` and `item` were stored. It restores those variables, computes `total_price`, and checks the balance guard. Before dispatching the second remote call it pushes a return address for `buy_item_step_3`. Here the live-variable analysis reduces the context payload: `amount` and `attr_1` are no longer live past this point and are not serialized, so only `item` and `total_price` are carried forward.

Note that entity type annotations are updated in the output: the parameter `item: Item` becomes `item: str`, because in the distributed setting an entity reference is represented by its key, whose type is determined from the `__key__` method of the entity class.

4.5 Compiling Loops and Branches

Acyclic code with remote calls maps cleanly onto CPS chains, because the splitting pass can process the statement list top-to-bottom and cut at each remote call site

without ever revisiting a statement. Control flow structures break this property: a loop body is executed an unknown number of times, and a conditional produces divergent successor statements whose live variables and continuation chains must be tracked independently. The compiler generalises the splitting algorithm to handle these cases by emitting one registered step function per control-flow join point, using the runtime's message dispatch as the sole mechanism for transferring control between them.

Conditional Branches

A conditional whose branches contain no remote calls is left in place: the splitting pass treats the entire `if` statement as a single straight-line unit and its successor statements remain in the same compiled function. When at least one branch contains a remote call, the post-`if` statements must be emitted as a continuation reachable from that branch. The pass therefore splits off the statements following the `if` into a new step function per branch that contains a remote call, allowing each branch to carry only the variables that are live along its own execution path into its respective context dictionary. Branches that contain no remote call are terminated inline with a `send_reply` or a direct dispatch to the appropriate successor, without pushing any continuation.

To illustrate, consider `conditional_restock`, a method that restocks an item by one unit only if a reference to the item is provided, and returns early otherwise:

```
def conditional_restock(self, item: Optional[Item] = None) -> str:
    if item is None:
        return "No item provided"
    item.update_stock(1)
    return "Restock complete"
```

LISTING 5: Original `conditional_restock`

The `if`-branch contains no remote call and terminates the method immediately, while the `else`-path issues a remote call to `update_stock` before reaching the final `return`. The compiler identifies that the post-`if` continuation is only reachable from the branch that performs a remote call. The early-return branch is therefore terminated inline with a `send_reply`, while the remote-call branch pushes a continuation for the remaining statement and dispatches to the item operator. This produces two step functions shown in Listing 6.

```python
@user_operator.register
async def conditional_restock(ctx: StatefulFunction, item: Optional[str] = None,
↪  reply_to: list = None) -> str:
    if item is None:
        # Branch with no remote call: reply immediately and terminate
        return send_reply(ctx, reply_to, "No item provided")

    # Branch with a remote call: push continuation and dispatch
    reply_to = push_continuation(ctx, reply_to, 'user',
                                 'conditional_restock_step_2',
                                 ctx.key, {})
    ctx.call_remote_async(operator_name='item',
                          function_name='update_stock',
                          key=item, params=(1, reply_to))

@user_operator.register
async def conditional_restock_step_2(ctx: StatefulFunction,
                                     func_context,
                                     placeholder_return=None,
                                     reply_to: list = None):
    # Shared post-if continuation: reached only via the remote-call branch
    return send_reply(ctx, reply_to, "Restock complete")
```

LISTING 6: Compiled output of `conditional_restock`

In this case, because neither branch defines any local variables that are live at the continuation, the context payload passed to `push_continuation` is empty. In general, any variable assigned before or within one branch and read after the `if` would appear in the live-variable set and be serialized.

While Loops

A loop whose body contains a remote call cannot be unrolled statically. The compiler instead reifies the loop as a *recursive step function*: a registered function that, when invoked, evaluates the loop condition and either dispatches to itself (to execute another iteration) or falls through to a separate post-loop step. Each iteration of the source-level loop corresponds to one asynchronous invocation of the loop-header step, and the loop condition is re-evaluated on each entry.

This transformation is an instance of the standard equivalence between iteration and tail recursion [42]: the loop-header step is a tail-recursive function whose back-edge is a tail call to itself and whose exit edge is a call to the post-loop continuation. Under CPS, this is the natural representation of loop control flow, as there is no primitive loop construct in the continuation-passing intermediate form and iteration is expressed entirely through tail calls [2]. The transformation preserves the loop's semantics provided the loop-carried state (the variables live at the loop header) is threaded through each invocation.

The structure produced for a loop is standardized. Given a loop in function `f`, the compiler decomposes it into three kinds of step function. A *loop-header step* checks the condition; if it holds, the step executes the body up to the first remote call or, if the body contains no remote call, dispatches back to itself directly; if it does not hold, the step dispatches to the post-loop step. A *post-loop step* contains the statements that follow the loop in the source. Finally, one or more *body-continuation steps* hold the code between successive remote calls inside the body and each terminate by dispatching back to the loop-header step to begin the next iteration. The set of variables live at the loop header (the loop's "carried state") is packaged into the context

dictionary at every dispatch into the header, so that accumulators and loop-scoped bindings are restored across iterations.

The `drain_stock` (see Section 4.1) method illustrates the pattern. Recall that after linearization the loop condition becomes: `while True: attr_2 = item.get_stock() ; if not (0 < attr_2 - 1): break; ...`, with the remote call hoisted to the top of the body so that it is reached at the start of every iteration. The loop-header step therefore begins with a `push_continuation` that registers the next step as the reply address for `get_stock`, then dispatches the call:

```
@user_operator.register
async def drain_stock_step_2(ctx, func_context, placeholder_return=None,
↪  reply_to=None):
    params = resolve_context(ctx, func_context)
    (item, total) = (params.get('item'), params.get('total'))
    reply_to = push_continuation(ctx, reply_to, 'user',
                                 'drain_stock_step_4', ctx.key,
                                 {'item': item, 'total': total})
    ctx.call_remote_async(operator_name = 'item', function_name = 'get_stock', key =
    ↪  item, params = (reply_to,))
```

The returned stock value is received by `drain_stock_step_4`, which evaluates the hoisted condition. If the condition does not hold, it dispatches to the post-loop step `drain_stock_step_3`; otherwise it issues the body's remote `update_stock` call with `drain_stock_step_5` as its continuation. The latter increments the accumulator and dispatches back to `drain_stock_step_2`, closing the loop. The accumulator `total` is live at the loop header and is therefore included in the context dictionary at each dispatch that targets it.

For Loops

For loops over collections are handled analogously, with one additional mechanism: because the loop-header step re-enters fresh on every iteration, the compiler cannot rely on Python's implicit iterator object to track position. It instead introduces an explicit integer loop-index variable, initialises it to zero before the loop, and increments it in the body. The loop-header step compares the index against the length of the iterable to decide whether to continue, and assigns the current element by subscripting the iterable on each iteration. The index and the iterable are both included in the loop's live set and are therefore carried through the context dictionary at every dispatch back to the header.

Nesting introduces no new concepts: inner loops use distinct indices and inner loop-header steps are emitted alongside outer ones. A body-continuation that follows a remote call inside the inner loop dispatches to the inner header; a body-continuation that follows the inner loop as a whole dispatches to the outer header. The live set grows with each level of nesting, as variables from all enclosing loops must be preserved.

Break and Continue

With the new generated functions we can also create mappings for `break` and `continue`, because each loop already exposes exactly the two dispatch targets these statements need: the post-loop step and the loop-header step. The compiler rewrites `break` as an unconditional dispatch to the enclosing loop's post-loop step, and `continue` as an unconditional dispatch to the enclosing loop's header step. In both cases the

current live-variable set is packaged into the context dictionary of the target step exactly as it would be for any other cross-step dispatch, so that any variables the target subsequently reads are correctly restored. Both statements also terminate the current compiled step, so statements written after them in the source are unreachable and are not emitted into the current function; they will only appear in the compiled output if they are reachable along some other control-flow path, in which case the splitting pass emits them as part of the appropriate successor step.

For nested loops the dispatch target is determined by the lexically enclosing loop, matching Python's semantics. The compiler tracks the loop context during the splitting pass and resolves each `break`/`continue` against the innermost enclosing loop's header and post-loop step identifiers.

Scaling to Complex Control Flow

The algorithm applies recursively to arbitrarily nested combinations of these constructs, with the live set at each join point computed as the union along every incoming path. The number of generated step functions therefore grows with control-flow complexity rather than simply with the remote-call count, but the output always mirrors the structure of the source.

4.6 Recursion

At a low level, recursion works because of the call stack. Every time a function is called, the runtime creates a stack frame containing the function parameters, local variables, a return address indicating where to continue once the call completes, and any saved intermediate state. When the recursive call returns, the frame is popped and execution resumes from the saved return address with the saved variables restored. This is precisely the structure that the reply-to stack replicates in the distributed setting: each step function serializes its live local variables into the context dictionary, pushes a return address record onto the reply-to stack, and dispatches outward, with the runtime unwinding the stack by invoking each saved address in turn as results propagate back up the chain. A recursive call therefore requires no special treatment from the compiler: it is simply a dispatch that targets the same operator and function name as the caller, with the reply-to stack acting as the distributed call stack, growing with each recursive invocation and unwinding naturally as each frame completes.

Consider `discounted_sum`, which prices a list of items recursively, applying a discount to any item above a given threshold:

```
def discounted_sum(self, items: list[Item], threshold: int) -> int:
    if not items:
        return 0
    price = items[0].get_price()
    rest = self.discounted_sum(items[1:], threshold)
    if price > threshold:
        return rest + int(price * 0.9)
    return rest + price
```

The compiler splits this into three step functions. The entry point handles the base case inline and otherwise calls `get_price` on the head of the list, pushing a continuation for `step_2` with `items` and `threshold` saved into the context. When the price arrives, `step_2` saves `price` into the context for `step_3` and dispatches `discounted_sum`

on the same `user` operator with the tail of the list, exactly as it would any other cross-entity call. When the recursive call eventually returns, `step_3` restores `price` and computes the final result.

```
@user_operator.register
async def discounted_sum(ctx: StatefulFunction, items: list[str], threshold: int,
↪  reply_to: list = None) -> int:
    if not items:
        return send_reply(ctx, reply_to, 0)
    attr_1 = items[0]
    reply_to = push_continuation(ctx, reply_to, 'user', 'discounted_sum_step_2',
    ↪  ctx.key, {'items': items, 'threshold': threshold})
    ctx.call_remote_async(operator_name='item', function_name='get_price',
    ↪  key=attr_1, params=(reply_to,))

@user_operator.register
async def discounted_sum_step_2(ctx: StatefulFunction, func_context, price=None,
↪  reply_to: list = None):
    params = resolve_context(ctx, func_context)
    (items, threshold) = (params.get('items'), params.get('threshold'))
    reply_to = push_continuation(ctx, reply_to, 'user', 'discounted_sum_step_3',
    ↪  ctx.key, {'price': price, 'threshold': threshold})
    ctx.call_remote_async(operator_name='user', function_name='discounted_sum',
    ↪  key=ctx.key, params=(items[1:], threshold, reply_to))

@user_operator.register
async def discounted_sum_step_3(ctx: StatefulFunction, func_context, rest=None,
↪  reply_to: list = None):
    # Context restoration - unique per stack frame
    params = resolve_context(ctx, func_context)
    (price, threshold) = (params.get('price'), params.get('threshold'))
    if price > threshold:
        return send_reply(ctx, reply_to, rest + int(price * 0.9))
    return send_reply(ctx, reply_to, rest + price)
```

The correctness of this pattern under failure is inherited from Styx: each recursive invocation is an independent transactional operator call with exactly-once semantics, so partial execution of any frame is rolled back on failure without corrupting the reply-to stack accumulated by enclosing frames.

4.7 Concurrent Execution

While the aforementioned transformations allow developers to write imperative code that safely executes in Styx's distributed environment, strictly sequential logic forfeits a significant performance advantage of distributed systems: the ability to parallelize asynchronous dispatches. To bridge this gap, Obol introduces explicit constructs for concurrent execution.

First, we allow users to issue asynchronous requests using the `send_async` marker, transforming them into a fire-and-forget pattern. To adhere to Obol's structured request-response lifecycle, any function invoked via a `send_async` chain, which breaks away from the main execution stack, suppresses its return value.

This is a deliberate restriction relative to Styx's low-level API. Working directly against that API, a developer can wire up an arbitrary graph of callbacks: one function dispatches several others, each of those dispatches more, and the final reply to the original caller is emitted by whichever function happens to complete the chain, with no requirement that a result ever flow back to the function that issued a given call. That freedom is precisely what forces the manual continuation tracking Obol

exists to eliminate, since the developer alone is responsible for ensuring exactly one reply is eventually produced and routed to the right place. Obol trades it away: every cross-entity call either returns its result to its caller or is explicitly fire-and-forget, and the compiler guarantees a single well-defined reply path. The result is less expressive than hand-written Styx but relieves the developer of tracking continuations by hand and enforces a structured execution flow that mirrors the imperative source.

However, this design introduces a measurable performance trade-off: compared to a hand-written callback graph, Obol's structured reply path may traverse intermediate frames on the way back. Two subsequent mechanisms mitigate this overhead by eliminating unnecessary routing hops and by enabling concurrent dispatch of independent operations. Together, they recover most of the performance advantage of hand-written routing, leaving only a small residual overhead that we quantify in Chapter 5.

Fan-out and synchronization. Fire-and-forget semantics alone are insufficient when a computation depends on the return values of multiple independent remote operations. Consider the following function:

```
def get_discounted_price(self, item: Item, coupon: Coupon) -> int:
    price = item.get_price()
    discount = coupon.get_discount()

    return max(price - discount, 0)
```

Under the default sequential transformation, Obol awaits the result of `item.get_price()` before dispatching the call to `coupon.get_discount()`. Because these are cross-entity calls distributed across different workers, each invocation incurs a network round-trip. Consequently, the resulting total latency is the sum of their individual network and execution delays: $T_{\text{item}} + T_{\text{coupon}}$.

Because the price of the item and the discount amount are independent, there is no need to block the second request while waiting for the first network response. By issuing both asynchronous messages concurrently, the optimal theoretical latency is bounded by the slower of the two round-trips:

$$\max(T_{\text{item}}, T_{\text{coupon}})$$

To achieve this, Obol introduces the *gather* construct, which enables developers to explicitly fan out independent asynchronous requests and await their collective resolution. Using this construct, the expression is rewritten to:

```
def get_discounted_price(self, item: Item, coupon: Coupon) -> int:
    price, discount = gather(item.get_price(), coupon.get_discount())

    discounted_price = price - discount
    return max(discounted_price, 0)
```

Obol's `gather` is designed as the distributed semantic equivalent of Python's standard `asyncio.gather`. From the developer's perspective, it behaves identically: it accepts multiple asynchronous invocations, executes them concurrently, and returns an ordered tuple of their results once all have completed.

In contrast to `asyncio.gather` which orchestrates local coroutines on a single-threaded event loop, the distributed nature of Styx requires Obol to translate this abstraction into a persistent, stateful synchronization barrier. Because the calling function's local execution frame is destroyed after dispatching the network requests,

the runtime must reliably track the arrival of multiple independent asynchronous responses across potentially different workers.

When the function splitter encounters a `gather` statement, it translates the single logical operation into a multi-step barrier pattern. The compiled output for the aforementioned method demonstrates this transformation:

```
@user_operator.register
async def get_discounted_price(ctx: StatefulFunction, item: str, coupon: str,
↪  reply_to: list = None) -> int:
    # 1. Initialize the synchronization barrier
    _gather_id = init_gather_barrier(ctx, total=2, saved={},
    ↪  parent_reply_to=reply_to)

    # 2. Dispatch both requests with unique tags
    _g_reply_0 = [{'op_name': 'user', 'fun': 'get_discounted_price_step_2',
                   'id': ctx.key, 'context': {'_g_barrier': _gather_id, '_g_tag':
                   ↪  0}}]
    ctx.call_remote_async(operator_name='item', function_name='get_price',
                          key=item, params=(_g_reply_0,))

    _g_reply_1 = [{'op_name': 'user', 'fun': 'get_discounted_price_step_2',
                   'id': ctx.key, 'context': {'_g_barrier': _gather_id, '_g_tag':
                   ↪  1}}]
    ctx.call_remote_async(operator_name='coupon', function_name='get_discount',
                          key=coupon, params=(_g_reply_1,))

@user_operator.register
async def get_discounted_price_step_2(ctx: StatefulFunction, func_context,
                                      _gather_partial=None, reply_to: list = None):
    # 3. Synchronize incoming responses
    barrier_id = func_context['_g_barrier']
    _g_tag = func_context['_g_tag']
    (is_complete, _g_results, saved, parent_reply_to) = update_gather_barrier(
        ctx, barrier_id, _g_tag, _gather_partial)

    if not is_complete:
        return

    # 4. Resume execution once all responses are collected
    reply_to = parent_reply_to
    price, discount = _g_results
    discounted_price = price - discount
    return send_reply(ctx, reply_to, max(discounted_price, 0))
```

The compiled output for `get_discounted_price` illustrates the three phases through which Obol translates a `gather` into a persistent synchronization barrier: initializing the barrier state, dispatching the remote calls concurrently, and synchronizing their responses before resuming execution.

- **Barrier Initialization:** The compiler calls a helper function which allocates a unique, persistent record in the operator's context store. This record tracks the expected number of responses (in this case 2), a dictionary to hold arriving results, and the variables from the lexical environment that must be preserved.
- **Concurrent Dispatch:** Instead of chaining the remote calls sequentially, the compiler emits consecutive instructions. All calls register the same continuation function but are assigned unique integer tags for the context restoration step.
- **Synchronization and Resumption:** As each remote worker finishes its computation, it asynchronously invokes the shared continuation step. The `update_gather_barrier`

function processes the incoming message, saving the partial result under its respective tag. If the barrier remains incomplete, the function simply returns, yielding the worker thread back to the runtime without blocking. Upon receiving the final expected response, the barrier resolves, clears its state from the persistent context, and returns the ordered tuple of results to the continuation, allowing the synchronous business logic to proceed.

If any fanned-out call raises a domain exception, the failure propagates through the runtime's transactional machinery just as a sequential call's would: the entire invocation chain, including the barrier state and any partial results already recorded, is rolled back under Styx's serializable transaction boundary. A gather therefore never resolves with a partial tuple; it either yields all results or the transaction aborts. This differs from `asyncio.gather` with `return_exceptions=True`, which Obol does not provide.

This mechanism extends to dynamic collections, enabling unpacking of comprehension expressions directly into `gather` (e.g., `gather(*[item.get_price() for item in items])`). By orchestrating these state machine transitions at compile time, Obol grants developers the performance benefits of concurrent asynchronous dispatch while preserving the familiar sequential programming model.

Distributed tail-call optimization. Obol also implements a distributed tail-call optimization. When a step function's final operation is returning the result of another remote invocation, routing the response back through the intermediate entity serves no functional purpose. Instead, the compiler forwards the original caller's reply-to address directly to the final invocation, allowing it to respond to the caller without the intermediate hop. For instance, consider a transaction involving three entities: a `User`, an `Order`, and an `Inventory`. If `Order.confirm()` performs its state mutations and concludes by returning the result of `Inventory.reserve_stock()`, a naive return path would require the response to travel from `Inventory` back to `Order` and then to `User` (`User` → `Order` → `Inventory` → `Order` → `User`). By forwarding the reply-to address, `Inventory` responds directly to `User` (`User` → `Order` → `Inventory` → `User`), eliminating one network hop per delegated call.

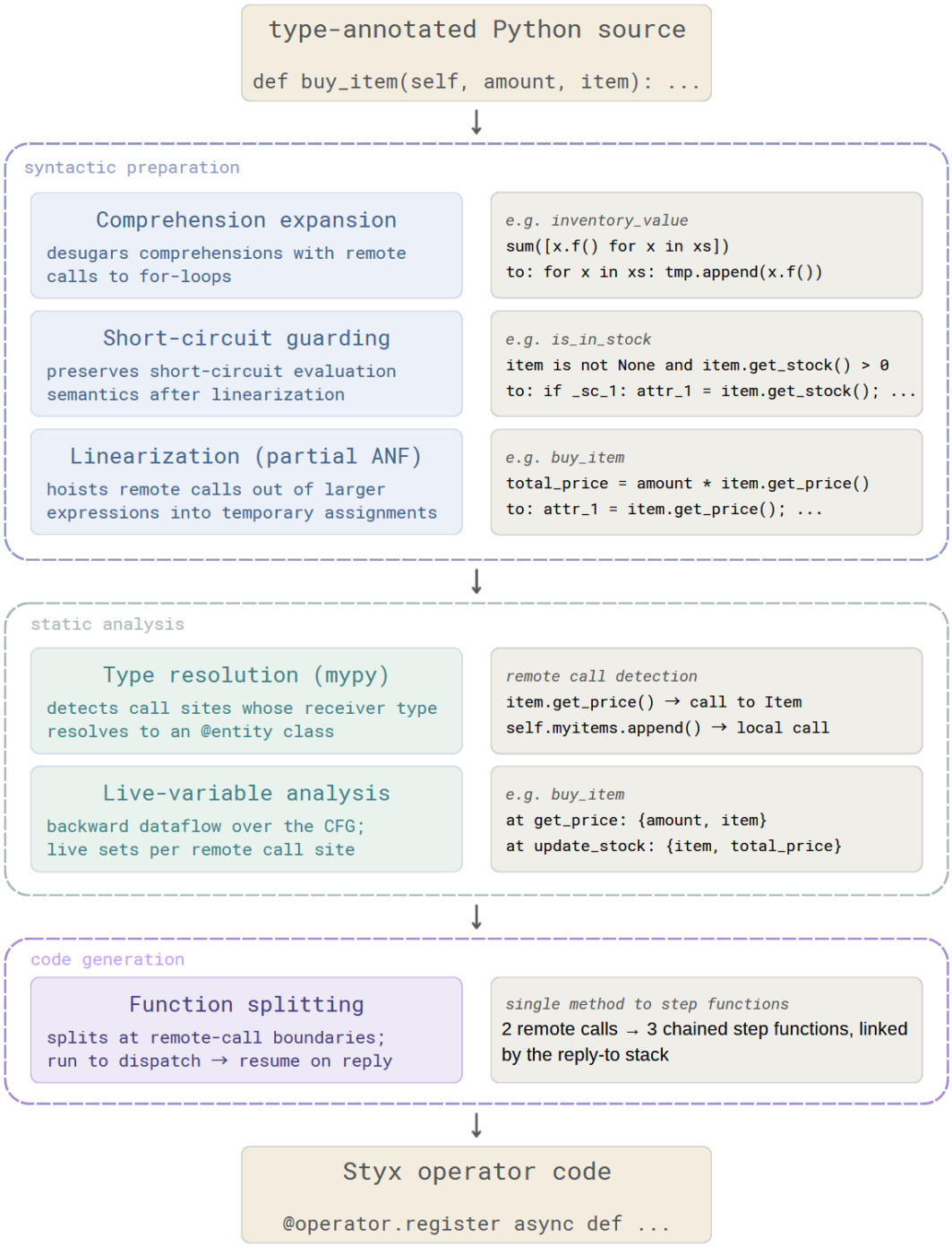

FIGURE 4.1: Overview of the Obol compilation pipeline. Type-annotated Python source passes through a syntactic preparation phase (comprehension expansion, short-circuit guarding, and linearization), a static analysis phase (`mypy`-based type resolution and live-variable analysis), and a final function-splitting pass that partitions each method at its remote-call boundaries, producing chained Styx operator functions linked by the reply-to stack.

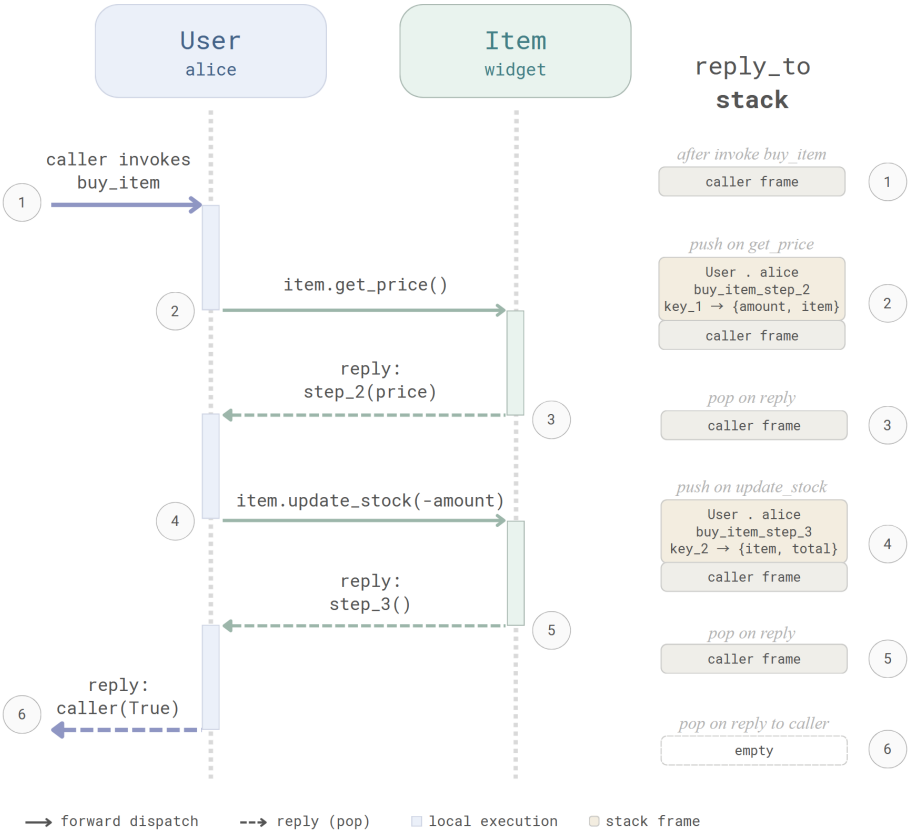

FIGURE 4.2: Runtime execution of the compiled `buy_item` transaction across the `User` and `Item` operators. Each forward dispatch pushes a continuation frame onto the `reply_to` stack and saves the live variables into the entity's context store; each reply pops the top frame and resumes the corresponding step function. The stack grows as remote calls are issued and unwinds as results propagate back to the original caller.

Chapter 5

Evaluation

The primary objective of Obol is to provide a high-level, imperative programming model for distributed workflows, one that keeps programs handling complex business logic maintainable, readable, and scalable. A natural question is what this abstraction costs. In this section we evaluate Obol along three axes: the semantic equivalence of the compiled code to its source, the reduction in code complexity it achieves, and the runtime overhead it introduces relative to hand-written, low-level Styx operator code. We assess semantic equivalence through randomized differential testing, and the other two using two standard database benchmarks, the Yahoo! Cloud Serving Benchmark (YCSB) [10] and the Transaction Processing Performance Council Benchmark C (TPC-C) [46]: code complexity by comparing the Obol and hand-written Styx implementations of each, and overhead by measuring their throughput. We find that compiled programs preserve the semantics of their source, and that the abstraction reduces developer effort substantially while remaining stable up to a saturation point close to that of hand-written code.

5.1 Correctness

Any comparison against hand-written operator code is meaningful only if the compiled program preserves the semantics of its source. Beyond a directed suite of hand-written comparative tests covering known-tricky constructs, we test semantic equivalence with randomized differential testing in the style of compiler fuzzing campaigns [48]: a seeded generator produces random sequences of entity operations, each sequence is executed both by the original Python classes and by the compiled operators on a live Styx deployment, and an oracle compares the two executions after every operation.

Methodology. Each sequence instantiates a twin world: a small population of `User`, `Item`, and `Coupon` entities created identically as in-memory Python objects and as Styx entities. The generator then draws operations from a weighted table covering 29 of our e-commerce application's entity methods, spanning every construct the compiler supports: straight-line transactions, loops with remote calls in their bodies and conditions, nested loops with `break` and `continue`, recursion, comprehensions over remote calls, short-circuit boolean expressions, cross-entity calls, and the concurrency constructs (`gather` over two and three heterogeneous entities, inside loops, and with dynamic fan-out over a comprehension). Argument generation is feedback-directed: generators inspect the live local state to bias inputs toward semantic boundaries—purchases of exactly the affordable amount, stock decrements of exactly the available stock, spending caps reached mid-iteration, iterables of mismatched lengths—which is where the control-flow splits introduced by compilation

are most likely to diverge. Every sequence is fully determined by its seed, so any failure reproduces from a single integer.

Oracle. The compiled program differs observably from its source in exactly the two ways described in Chapter 3: entity references are objects locally but routing keys on the wire, and domain exceptions raised locally are returned as error strings by the runtime. The oracle bridges both through normalization: entity objects and key strings are mapped to common canonical tokens (including entity-keyed dictionaries and entity-valued collections), and a local exception is required to surface as its message string, verifying that error propagation survives arbitrarily deep compiled call chains. After every operation the oracle additionally compares the complete observable state of every live entity, which detects divergences in effects even when return values agree, such as a continuation restoring a stale variable or a lost update in a loop iteration.

One subtlety makes a naive reference model unsound: Styx executes each invocation chain as a serializable transaction, so a domain exception rolls back all of the chain's state mutations, whereas plain Python does not. The reference therefore snapshots the entity world before each operation and restores it when a domain exception escapes, modelling the source program *under transactional-abort semantics* which the compiler must preserve. The campaign consequently exercises not only the translation of control flow but the inherited rollback guarantee.

Results. We executed 200 sequences of 50 operations each (10,000 operations in total). The full campaign passes: all return values, propagated error messages, and per-operation state snapshots agree between the source and compiled executions.

Scope. The campaign is testing the compilation, not the runtime: operations are issued sequentially, so serializability under concurrent contention and exactly-once delivery under worker failure are inherited from Styx and evaluated in its own work [32]. Fire-and-forget operations (`send_async`) are excluded: the construct is a compile-time marker whose argument must not be evaluated locally, so the reference interpreter cannot execute it faithfully, and its suppressed return value and racing effects leave no deterministic oracle in any case.

5.2 Developer Effort

Before reporting performance, we quantify the reduction in code that motivates Obol. The metric is a proxy for the manual continuation-management burden described in Section 2.2: the number of functions a developer must author and name, and the lines of code they must write, to express each transaction. It is a measure of developer effort, not a runtime result, and we keep that distinction explicit throughout.

We compare the Obol source a developer writes against hand-written Styx operator code produced by an author of the Styx system [32] for the same workloads, the New-Order and Payment transactions from TPC-C and the transfer transaction from YCSB. Both programs run on the same Styx runtime, so every difference reported here is attributable to the abstraction. The hand-written baseline is idiomatic Styx, written as a competent developer would write it, it is the code we benchmark against in the remainder of this section.

What we exclude. We report but do not count two categories of code, because they are common to any data model and are not where the stack-ripping burden lies. The first is schema and identity boilerplate: in Obol this is the `__init__` and `__key__` methods that declare each entity's fields and routing key; in the hand-written version the equivalent information is carried implicitly inside the dictionary literals and composite-key strings each operator constructs. The second is trivial `insert` operators in the Styx baseline, each a one-line `ctx.put`, which Obol subsumes into entity construction. Excluding both isolates the code that actually expresses transaction logic and cross-entity control flow.

Counting methodology. For each transaction we count the functions that participate in its execution and their logical lines of code, where a logical line is any non-blank, non-comment source line. On the Obol side these are the entity methods the transaction invokes, transitively; on the Styx side they are the registered operator functions together with the hand-written helper functions that reassemble results across the asynchronous reply handlers. Counts were produced by static analysis of the source.

Results. Table 5.1 reports the comparison. For the two TPC-C transactions, the hand-written implementation requires 22 functions where Obol requires 12, and 440 logical lines where Obol requires 246, a 1.8× reduction on both axes. For the YCSB transfer transaction the difference is smaller, since it involves no complex control flow and only a single cross-entity call whose result is not consumed downstream. Both implementations use two functions, and the gap is confined to lines of code (14 versus 9).

TABLE 5.1: Developer effort per transaction: functions (Fns) authored and logical lines of code (LLOC), excluding schema/identity boilerplate and trivial `insert` operators. Lower is better.

	Obol		Hand-written Styx	
Transaction	**Fns**	**LLOC**	**Fns**	**LLOC**
YCSB Transfer	2	9	2	14
TPC-C New-Order	6	149	12	272
TPC-C Payment	6	97	10	168

The function-count gap measures the stack-ripping burden directly. Every cross-entity interaction whose result is needed downstream forces the hand-written logic to split at the remote call: the dispatching operator cannot observe the reply, so the continuation becomes a separate registered function and the result is routed back to a coordinator that accumulates partial responses. The New-Order coordinator alone is made up of four such reply handlers (`get_warehouse`, `get_district`, `get_customer`, `get_item_with_stock`) plus two helpers that detect completion and pack the result, none corresponding to anything in the domain logic. In Obol the same transaction is a single method whose `gather` expresses the fan-out directly, and the compiler synthesizes the equivalent barrier and handlers. These numbers are therefore a lower bound on the benefit, not an average. The savings track the distributed control flow a transaction contains: transfer (YCSB), with no result aggregation, shows a small gap, while New-Order (TPC-C), which fans out and threads results through later

logic, shows a large one. For the complex business logic that motivates the abstraction—nested control flow, loops over remote calls, results combined across many entities—each additional remote call whose value is consumed adds another hand-written continuation, widening the gap.

5.3 Experimental Setup

All experiments were conducted on a single machine with two AMD EPYC 7413 24-core processors (providing 96 logical cores in total) and 503 GB of RAM. The full dataset fits entirely within memory across all configurations. Each Styx worker is deployed within a separate Docker container. To ensure that performance variances reflect compilation overhead alone, the system configuration was held identical across all evaluations, using 8 workers, 10 threads per worker, and a total of 80 state partitions. Each transaction executes as a single Styx invocation chain under the runtime's default serializable, exactly-once configuration. Finally, each benchmark run included a 60-second warmup period to stabilize performance metrics before recording results.

Workload generation. We drive each system with an open-loop load generator that submits transactions at a fixed input rate, sweeping the offered rate upward from 100 txn/s until each system saturates; the range therefore differs across workloads and is given by the extent of each figure's x-axis. While the system keeps up, achieved throughput tracks the offered rate and latency stays flat; once the offered rate exceeds what the system can sustain, throughput plateaus and latency grows without bound as requests queue. We report end-to-end latency at the median (p50) and tail (p99) against offered throughput, with latency on a logarithmic axis, and characterize each system by two quantities. The first is its stability below saturation: how closely its latency tracks the hand-written baseline while both keep up with the offered load. The second is its saturation point, which we define as the offered rate at which p50 latency first crosses 1 second (1000 ms), marking the transition from the flat, healthy regime into the overloaded regime where requests queue and latency grows without bound. The saturation point is therefore the maximum throughput a system sustains at acceptable latency.

Baselines. We compare three systems on the Styx runtime:

- **Hand-written Styx** is the expert-authored, low-level operator code described in Section 5.2. It is idiomatic Styx and represents the performance ceiling we measure the abstraction against.
- **Obol (gather)** is our compiler's output from the sequential entity source, with independent cross-entity reads expressed through the `gather` construct (Section 3.3) so that they fan out concurrently.
- **Obol (no gather)** is the output from the same source written without `gather`, so that every cross-entity call is dispatched sequentially and each reply is awaited before the next is issued. This isolates the cost of expressing a transaction's independent calls as a strictly sequential chain.

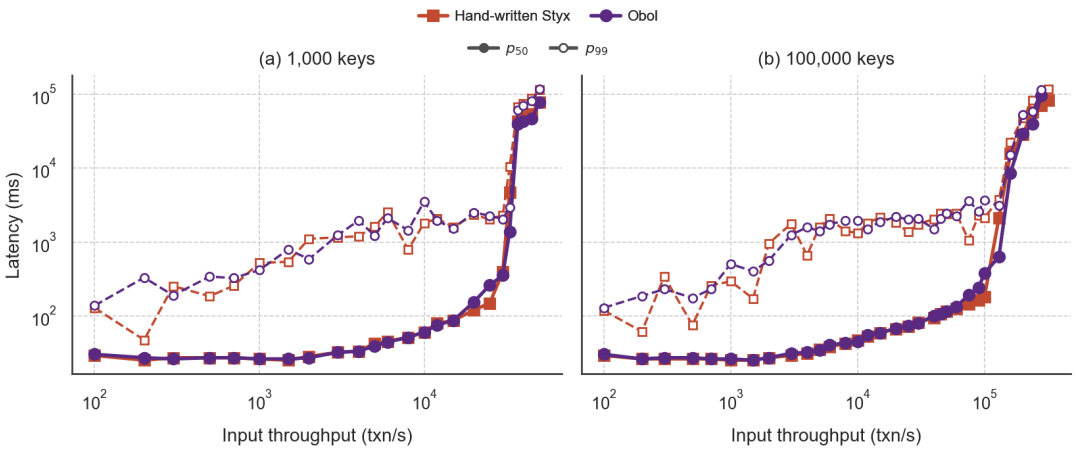

FIGURE 5.1: YCSB transfer saturation curves: offered throughput against end-to-end latency (log axis), p50 in solid lines, p99 in dashed lines with hollow markers. Panel (a) high-contention (1,000 keys), panel (b) low-contention (100,000 keys). The transfer transaction issues a single remote call, so one Obol curve covers both `gather` and no-`gather`.

5.4 YCSB

YCSB's transfer transaction moves a fixed amount between two accounts. In Obol it is a single entity method that debits one account and issues one remote call to credit the other; the hand-written baseline is the corresponding pair of operator functions. Because the transaction issues exactly one remote call, it exposes no independent dispatches that could be overlapped: the `gather` and no-`gather` compilations coincide, and the only quantity that can differ between Obol and hand-written Styx is the raw cost of the compiled continuation machinery. Transfer therefore serves as a control. It isolates compilation overhead from the dispatch-scheduling effects that dominate the TPC-C comparison (Section 5.5), and establishes the floor against which those results are read.

As we will with TPC-C, we sweep the offered load under two configurations that vary contention through the size of the key space. With **1,000 keys** the load is concentrated, so concurrent transfers frequently contend for the same accounts, with **100,000 keys** the same load is spread across a key space two orders of magnitude larger and conflicts are rare. Figure 5.1 reports both.

Results. In both regimes the two p50 curves are indistinguishable across the entire pre-saturation range. From the flat, lightly loaded region near 30 ms up through the onset of queueing, Obol tracks hand-written Styx to within measurement noise, and the two cross the 1 s saturation threshold at effectively the same offered rate: near 3×10^4 txn/s under high contention (1,000 keys) and near 1.4×10^5 txn/s under low contention (100,000 keys). Tail latency behaves the same way: both systems exhibit the noisy p99 characteristic of an open-loop generator below saturation, with no systematic separation between them.

This is the most conservative possible test of the abstraction. Transfer offers nothing to optimize: a single remote call cannot be fanned out, so Obol's output is a straight CPS chain with the same number of network hops as the hand-written code. That the curves overlap up to nearly the same saturation point establishes that compilation in isolation costs essentially nothing, that compiled $\approx$ hand-written when

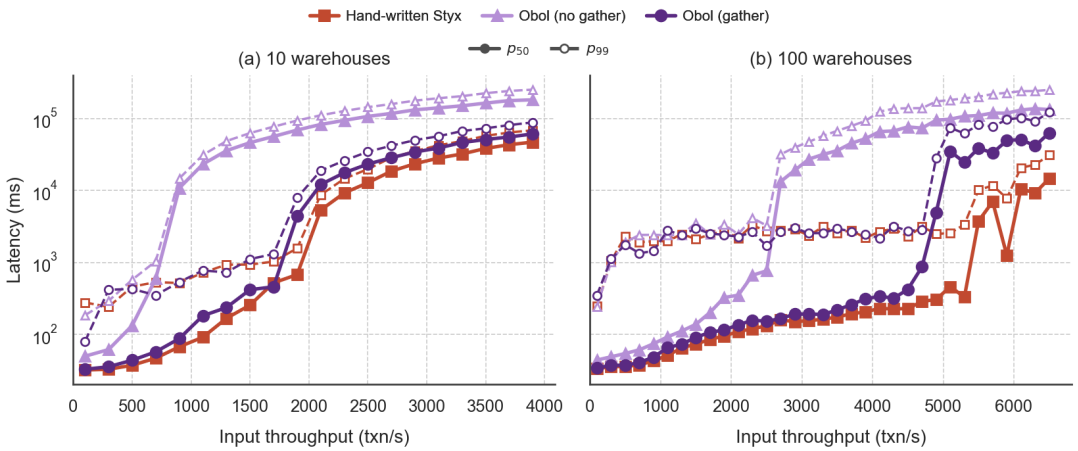

FIGURE 5.2: TPC-C saturation curves for two warehouse configurations: offered throughput (x-axis) against end-to-end latency on a logarithmic axis (y-axis), p50 in solid lines, p99 in dashed lines with hollow markers. Panel (a) is high-contention (10 warehouses), panel (b) low-contention (100 warehouses).

the only variable is the generated machinery. This is what licenses the reading of the TPC-C results in Section 5.5: the gap that opens for Obol (no `gather`) there cannot be attributed to compilation overhead, since that overhead is ≈0 here, and must instead be the cost of serializing independent calls that `gather` is able to overlap.

5.5 TPC-C

TPC-C [46] is the standard online-transaction-processing benchmark and stresses the parts of the programming model that matter most: cross-entity transactions with branching, loops over line items, and result aggregation across several entities. We implement its two main transactions, New-Order and Payment, issued in an even mix over the standard warehouse–district–customer–item–stock schema, expressed both as Obol entities and as equivalent hand-written operators.

We again run the sweep under two warehouse configurations that place the system in opposite contention regimes. With **100 warehouses** the key space is large and conflicts between concurrent transactions are rare, measuring throughput under low contention. With **10 warehouses** the same load is concentrated onto a tenth of the keys, so transactions contend heavily for the same warehouse and district records, and the time each transaction holds those hot keys becomes the dominant factor. Figure 5.2 reports both.

Results. Below saturation, Obol (gather) and hand-written Styx track one another closely, with Obol (gather) holding p50 within a few percent of hand-written latency in both settings, confirming that the compiled code introduces no instability of its own. Obol (no gather) runs at a somewhat higher latency throughout and saturates earlier, as expected from dispatching its independent calls sequentially rather than concurrently. The three also differ in where they leave the flat, healthy operating range. Under high contention (Figure 5.2a) hand-written Styx crosses 1 second at roughly 1,900 txn/s and Obol (gather) at roughly 1,700 txn/s, about 200 txn/s earlier. Under low contention (Figure 5.2b), where the larger key space lets both reach far higher absolute throughput, hand-written Styx saturates near 5,500 txn/s and Obol

(gather) near 4,700 txn/s, a gap of roughly 800 txn/s. The compiled fan-out barrier therefore reproduces the stability of expert-written reply handlers and saturates only modestly earlier, with the absolute gap widening at the higher throughputs that low contention permits.

The dominant gap is between the two Obol variants. Obol (no gather) saturates far earlier in both settings, near 750 txn/s with 10 warehouses and near 2,400 txn/s with 100 warehouses, beyond which its latency climbs by orders of magnitude. The penalty is the price of serializing calls that could have run concurrently, not the price of compilation, and it dwarfs the small gap between Obol (gather) and hand-written Styx.

Takeaway. Two findings separate cleanly. The compiled code is stable: when the source expresses the available concurrency, Obol (gather) tracks hand-written latency to within a few percent up to saturation and saturates only modestly earlier. What determines saturation is whether independent calls overlap: `gather` recovers nearly all of hand-written Styx's throughput, while forgoing it collapses it. The small residual gap between Obol (gather) and hand-written Styx — roughly 200 txn/s under high contention and 800 under low — is the price of the programming model rather than of compilation: the hand-written version exploits reply wiring that Obol's structured call-and-return semantics deliberately do not express (Section 4.7), and the tail-call and gather optimizations recover most, though not all, of that routing freedom.

Chapter 6

Limitations and Future Work

Obol is a research prototype that automates a non-trivial source-to-source transformation, and the current implementation accepts a restricted subset of Python. A design goal is that any construct the compiler cannot accurately translate to the distributed setting is rejected at compile time, rather than silently compiled to code with different behaviour. The prototype approximates this goal: most of the restrictions described below are enforced by static checks.

6.1 Unsupported Python Constructs

Several Python language features are rejected by the compiler. Lambda expressions and nested function or class definitions cannot appear inside entity methods, as the CPS transformation requires every continuation to be a named, registered step function. Generator expressions containing remote calls and `yield` statements are not supported. The `else` clause on loops, context managers (`with`), the walrus operator, `match`/`case`, and user-defined decorators on entity methods are not currently supported. The `async`/`await` keywords are intentionally absent from the input model, as asynchrony is expressed at the call site rather than the definition.

Exception handling is handled only when local to a single step. A `try`/`except` spanning a remote call would require the compiler to split the handler into a continuation reached only on the exceptional reply path, which in turn requires the runtime to deliver exceptions as a distinct reply variant routed back up the reply-to stack to the frame whose `try` block encloses the call. More fundamentally, it conflicts with the present transactional model: a domain exception currently aborts the entire invocation chain, since the chain is one serializable transaction. Catching an exception mid-chain would instead require the rollback to be scoped to the failed sub-call while the enclosing transaction proceeds - that is, nested transaction boundaries or savepoints, which Styx does not currently expose. Supporting cross-boundary exception handling is therefore not only a code-generation extension but a runtime one. Finally, inheritance between entity classes is not modelled; each class is treated independently. Most of these restrictions are engineering limitations rather than fundamental constraints of the approach, and lifting them is left to future work.

6.2 Aliasing Across Remote Calls

Obol serializes the live locals captured at each split point with `pickle` [33], and persists entity state separately through the runtime's `ctx.put`/`ctx.get` interface. Pickle preserves shared structure within a single serialization: if two captured locals reference the same underlying object, that sharing is reconstructed when the continuation

restores them, so ordinary non-entity objects cross a step boundary with their internal aliasing intact. What does not survive is an alias between a local variable and an entity's state. A continuation restores its locals from the captured context and re-reads the entity's state through an independent `ctx.get`, so the two are reconstructed as distinct objects even when the source created them as aliases. A reference taken into the state before a remote call is therefore detached after it: mutations through the reference do not reach the persisted state, and the subsequent `ctx.put` would in any case overwrite them.

```
def restock_and_record(self, item: Item) -> None:
    items = self.myitems    # local alias into entity state
    item.update_stock(1)    # remote call: splits the method here
    items.append(item)      # intended to update self.myitems
```

Here `items` aliases `self.myitems` when the method begins, but the `update_stock` call splits the method at this point. In the continuation, `items` is restored from the captured context while `self.myitems` comes from a new state read, so the `append` mutates a detached copy and the recorded purchase is lost. The mutation must instead be written against the state directly, as `self.myitems.append(item)`, within the step that performs it.

This is not merely an artifact of serializing locals and state through different channels. Because each step reads the entity's state afresh at its own entry, a continuation cannot hold a live reference into the state across a suspension point: while the method is suspended awaiting a reply, the authoritative state is the entity's own, which any step observes only by re-reading it, not a snapshot captured in an earlier step. The restriction this imposes is that mutable entity state must be reached through `self` within each step rather than through a local alias threaded across a remote call.

6.3 Static Analysis Requirements

The compiler relies on static type information produced by `mypy` to identify which call sites target remote entities. Because Python's type system is optional, the compiler requires that every entity method parameter and return value carry an explicit type annotation, and rejects programs in which any annotation is missing or cannot be resolved to a concrete type. This is stricter than idiomatic Python but is necessary for the compiler to statically determine the full message-passing structure of the output program. For the same reason, all entity class definitions must be visible in a single compilation unit, and entity references constructed dynamically cannot be resolved statically and are rejected. The `get_entity_by_key` helper is a compile-time marker rather than a runtime function and may only appear in contexts where the target entity type is statically known.

6.4 Single Runtime Target

Obol currently targets Styx exclusively. The generated code uses Styx-specific primitives (`ctx.get`, `ctx.put`, `ctx.call_remote_async`) and registers its step functions with Styx `Operator` objects. Retargeting another SFaaS runtime would require adapting this syntax and reviewing the assumptions the prototype inherits from Styx, in particular the exactly-once transactional boundary around a single invocation chain and the partitioned state model. This is a natural extension rather than a fundamental restriction of the approach.

Chapter 7

Related Work

Obol is part of a broader set of systems designed to bridge the gap between familiar developer abstractions and the complexities of distributed execution.

Workflow orchestrators. Building non-trivial serverless applications typically requires composing multiple functions into larger workflows. For long-running, multi-step business logic, this composition is often expressed through dedicated workflow orchestrators. Systems like AWS Step Functions [3] and Google Cloud Workflows [18] define workflows declaratively as JSON or YAML state machines executed by a standalone, separately billed cloud service. Apache Airflow [1] provides a programmatic Python interface for task definition, but it retains the same decoupled scheduling architecture. Similarly, gg [16] utilizes a long-running coordinator to dispatch work to short-lived workers, and Kappa [50] addresses the related problem of executing long-running applications across function-timeout boundaries by checkpointing state through a central coordinator. These systems have proved successful for coarse-grained data pipelines, where tasks are minutes-long batch jobs and inter-task state is naturally externalized to object storage or a data warehouse. They are a poor fit for the workloads Obol targets. The unit of work is a task rather than a method call, so transactional cross-entity interactions cannot be expressed within a single task. The workflow is a separate artifact from the application code, breaking the locality of reasoning that object-oriented programming provides. Obol stays inside the application: the workflow is the method body, and the compiler is responsible for distributing it.

A different approach to orchestration relies on *deterministic replay* rather than compilation. Azure Durable Functions [5] and Temporal [44], discussed in Section 2.3, record every external interaction of a workflow to a persistent event history. Upon failure, the runtime re-executes the program from the beginning and returns the recorded result instead of repeating the interaction. For this mechanism to be safe, the workflow code must be strictly deterministic: it cannot read the system clock, generate random numbers, spawn native threads, or perform I/O directly. Developers must instead route operations through runtime-provided library wrappers. Obol achieves durability from the opposite direction. Synchronous Python is compiled into a message-passing call-graph executed by the Styx runtime, inheriting exactly-once semantics from asynchronous barrier snapshotting [7]. The restrictions Obol imposes—such as mandatory type annotations and statically resolvable entity references—are purely constraints on static analysis, not limitations on runtime replay.

Compiling stateful entities to dataflow. Obol is most directly indebted to Stateful Entities [51], from which it inherits its central premise, that distributed stateful workflows can be written as ordinary object-oriented Python and compiled to

dataflow, and its core function-splitting technique. Our contribution is not the idea of this compilation but a redesign of how it is carried out. Where Stateful Entities is deliberately runtime-agnostic, compiling to a portable intermediate representation that runs on several dataflow engines, Obol makes the opposite choice and targets a single runtime, Styx, exclusively. Committing to one runtime is what makes the redesign possible and is what distinguishes Obol along three axes. First, guarantees: by generating code directly against Styx rather than a lowest-common-denominator IR, Obol preserves serializable, exactly-once transactions across an entire cross-entity call-graph, where the runtime-agnostic approach inherits only the message-level processing guarantees of whichever engine it targets. Second, coordination: Stateful Entities routes cross-entity calls by traversing a centralized execution graph anchored at the originating operator, whereas Obol threads a decentralized reply-to stack of defunctionalized continuations through the messages themselves, with no central point of control. Third, expressiveness: because Stateful Entities resolves nested calls by statically merging execution graphs, it cannot represent a self-referential call, so recursion falls outside its model, and it dispatches independent cross-entity calls strictly sequentially; Obol supports recursion directly, since the reply-to stack is the distributed call stack, and exposes explicit constructs for concurrent fan-out.

Closer in technique to Obol is Unum [27], which replaces centralized orchestration with compiler-generated, decentralized coordination implemented via per-function logic and shared storage. Unum and Obol share the foundational insight that distributed continuations are best handled as a compiler problem rather than a manual burden placed on the developer. However, they target opposite ends of the FaaS design space. Unum coordinates stateless functions whose state resides in external storage, whereas Obol assumes a runtime that co-locates state with compute, leveraging the transactional guarantees of the underlying dataflow engine.

Actor systems. The entity abstraction Obol exposes is closely related to the virtual actor model of Orleans [6], where addressable stateful objects are transparently placed and activated by the runtime. Obol differs in compiling sequential method bodies that span multiple entities into a single transactional call-graph, rather than treating each actor invocation as an independent message.

Choreographic programming. Choreographic programming [28, 11] lets developers describe a distributed system's interactions as a single unified program, which a compiler then decomposes into individual per-node programs through a process called endpoint projection. Choral [17] embeds this idea in an object-oriented language, expressing choreographies as generic classes parameterised over roles, while Pirouette [20] provides a higher-order typed variant with mechanised correctness guarantees. HasChor [37] takes a functional approach, embedding choreographies as a library abstraction in Haskell and enabling endpoint projection through standard functional combinators.

The assumption underlying all of these systems, however, is that participants are stateless: locations exchange values but hold no persistent state, and there is no transactional model governing their interactions. Furthermore, choreographic programming targets a holistic, system-wide view of communication, describing the entire interaction protocol among all participants from a bird's-eye perspective. Obol's scope is narrower and more local: it concerns itself not with specifying cross-service protocols but with compiling the internal logic of individual stateful entities, each

of which may call into its peers as part of carrying out a transaction. The resulting structure is closer to a collection of co-operating microservices than to a centrally choreographed ensemble.

Chapter 8

Conclusion

Distributed stateful workflows impose a significant burden on developers: what reads as straightforward sequential logic must be manually decomposed into chains of asynchronous callbacks, with state captured, serialized, and routed by hand, diverting developer attention away from business logic. This paper presented Obol, a compiler-driven approach that eliminates this burden by allowing developers to express distributed workflows as ordinary sequential, object-oriented code, and automatically transforming it into the asynchronous message-passing form required by the Styx SFaaS runtime.

The core of Obol is a multi-stage compilation pipeline grounded in continuation-passing style and defunctionalization. Starting from type-annotated source code, the pipeline applies a series of preparatory transformations before splitting functions at remote call boundaries, producing chains of registered step functions connected by an explicit distributed stack. The compiler handles a wide range of control flow constructs and provides explicit primitives for expressing concurrent dispatch patterns that exploit Styx's asynchronous architecture, alongside optimizations that eliminate unnecessary intermediate network hops.

Crucially, this abstraction introduces little to no transactional overhead. The generated code executes directly within Styx's dataflow engine and inherits its serializable, exactly-once guarantees without modification, producing an output structure that closely mirrors what a developer would write by hand against the low-level API. Our evaluation on YCSB and TPC-C confirms this. On a workload with no exploitable concurrency (YCSB transfer) the compiled code is indistinguishable from hand-written Styx, isolating compilation overhead at essentially zero; on TPC-C, once available concurrency is expressed, compiled code matches hand-written throughput, staying within a few percent of hand-written latency up to saturation and saturating only modestly earlier, by roughly 200 txn/s under high contention and 800 txn/s under low contention.

By raising the level of abstraction for SFaaS programming, Obol improves the readability, maintainability, and scalability of distributed stateful applications, allowing teams to reason about complex workflows in familiar terms while the compiler bears the cost of distributed coordination. Obol demonstrates that the stack-ripping problem in distributed systems is not a fundamental constraint of the programming model, but a consequence of working at the wrong level of abstraction. A sufficiently structured compiler can close the gap between how developers naturally reason about stateful interactions and how those interactions must be expressed at the runtime level, without sacrificing the performance or correctness properties that motivate the underlying system.

Appendix A

Reflection

The Data Science and Artificial Intelligence Technology (DSAIT) field is concerned not only with data-driven and intelligent systems themselves but with the infrastructure that makes such systems scalable, reliable, and tractable to build. A persistent theme at this infrastructure layer is the tension between performance and programmability: the distributed runtimes that deliver throughput and strong execution guarantees tend to expose low-level interfaces that are awkward to program against directly. Much of the field's recent trajectory can be read as an effort to close this gap by raising the level of abstraction, from declarative query languages layered over dataflow engines, to dataframe and high-level streaming APIs, but also to compiler-based approaches that let developers express intent in a familiar paradigm while the system synthesizes an efficient distributed execution plan. This thesis is positioned within that last category: it applies established programming-languages techniques — continuation-passing style, defunctionalization, and live-variable dataflow analysis — to a data-systems problem, the manual decomposition of transactional cross-entity workflows, letting developers write distributed stateful logic as ordinary sequential code that a compiler then lowers to the message-passing form the runtime requires.

In doing so, the work draws on two research traditions that DSAIT increasingly brings into contact. From the data-systems side, it builds on the stateful serverless paradigm and its principle of co-locating state with compute, inheriting the transactional and exactly-once guarantees that runtimes such as Styx obtain from asynchronous barrier snapshotting. From the programming-languages side, it reuses classical results on compiling sequential control flow into explicit message-passing form. The contribution is therefore one of synthesis rather than of a single new primitive, and it reflects a broader convergence in the field, in which the boundary between a database and a programming language is blurring and the application program itself becomes something a compiler can transform — much as a query optimizer rewrites declarative SQL — into a distributed execution plan.

Positioning the work against current industry trends is equally instructive. The dominant approaches to distributed workflows today are external orchestration (AWS Step Functions, Airflow) and durable execution via deterministic replay (Temporal, Azure Durable Functions), and the latter in particular is rapidly becoming the default programming model for stateful serverless applications. This thesis deliberately diverges from that direction: rather than achieving durability through replay, which constrains developers to a deterministic dialect of their language, it compiles unrestricted sequential code against a runtime that provides durability natively, preserving serializable cross-entity transactions that the replay-based model does not express. The work is thus best understood as an exploration of an alternative point in a design space the field is otherwise consolidating around, and its evaluation

adopts the empirical conventions of both the systems and the PL communities to make that comparison rigorous.

The relevance of this line of work is likely to grow rather than diminish. The same stateful dataflow substrate that Obol targets increasingly underpins data-intensive and AI-driven applications: feature stores, real-time inference pipelines, and the orchestration of long-running, stateful agentic backends all recreate the stack-ripping problem at the application layer. As these systems become more interactive and more stateful, the ability to write distributed coordination logic as ordinary sequential code, without surrendering transactional correctness, addresses a need that extends well beyond our current scope.

Appendix B

Use of Generative AI Tools

The AI tools used in this work are the following: Google Gemini, Anthropic's Claude, and OpenAI's ChatGPT. Their use was strictly supportive, and I retain full intellectual and creative responsibility for all ideas, arguments, code, and text in this thesis. All AI-assisted outputs were carefully reviewed, verified for accuracy, and checked for originality before inclusion.

These tools were used in several ways. During the research phase, they assisted in summarizing relevant literature, identifying papers for the related work section, and exploring possible implementations and validations of the transformations used in the code, as well as potential optimizations. During development, they were used to help identify bugs, generate test cases, and support code refactoring; however, the core contributions and implementation decisions remain my own. During the writing phase, they were used to improve the clarity of drafted text and to assist with LaTeX formatting.

All suggestions produced by these tools were critically evaluated, tested, or verified prior to use. No sensitive or confidential information was entered into any of these tools.

Bibliography

[1] Apache Airflow. 2015. URL: https://airflow.apache.org/.

[2] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992. ISBN: 978-0521416955.

[3] AWS Step Functions. 2016. URL: https://aws.amazon.com/step-functions/.

[4] Gavin Bierman et al. "Pause 'n' Play: Formalizing Asynchronous C#". In: *ECOOP 2012 – Object-Oriented Programming*. Ed. by James Noble. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 233–257. ISBN: 978-3-642-31057-7.

[5] Sebastian Burckhardt et al. "Durable functions: semantics for stateful serverless". In: *Proc. ACM Program. Lang.* 5.OOPSLA (Oct. 2021). DOI: 10.1145/3485510. URL: https://doi.org/10.1145/3485510.

[6] Sergey Bykov et al. "Orleans: Cloud Computing for Everyone". In: *Proceedings of the ACM Symposium on Cloud Computing (SoCC '11)*. Association for Computing Machinery, 2011, pp. 16–30. DOI: 10.1145/2038916.2038932.

[7] Paris Carbone et al. *Lightweight Asynchronous Snapshots for Distributed Dataflows*. 2015. arXiv: 1506.08603 [cs.DC]. URL: https://arxiv.org/abs/1506.08603.

[8] K. Mani Chandy and Leslie Lamport. "Distributed snapshots: determining global states of distributed systems". In: *ACM Trans. Comput. Syst.* 3.1 (Feb. 1985), pp. 63–75. ISSN: 0734-2071. DOI: 10.1145/214451.214456. URL: https://doi.org/10.1145/214451.214456.

[9] Chaoyi Cheng et al. "Developer's Responsibility or Database's Responsibility? Rethinking Concurrency Control in Databases". In: *13th Annual Conference on Innovative Data Systems Research (CIDR '23)*. Amsterdam, The Netherlands, 2023. URL: https://par.nsf.gov/biblio/10431671.

[10] Brian F. Cooper et al. "Benchmarking cloud serving systems with YCSB". In: *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*. New York, NY, USA: ACM, 2010, pp. 143–154. DOI: 10.1145/1807128.1807152.

[11] Luís Cruz-Filipe and Fabrizio Montesi. "A Core Model for Choreographic Programming". In: *Formal Aspects of Component Software*. Ed. by Olga Kouchnarenko and Ramtin Khosravi. Cham: Springer International Publishing, 2017, pp. 17–35. ISBN: 978-3-319-57666-4.

[12] Olivier Danvy and Lasse R. Nielsen. "Defunctionalization at Work". In: *Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP)*. Association for Computing Machinery, 2001, pp. 162–174. DOI: 10.1145/773184.773202.

[13] Datadog. *The State of Serverless*. Tech. rep. Accessed: 2026-02-15. Datadog, Inc., 2025. URL: https://www.datadoghq.com/state-of-serverless/.

[14] Adam Dunkels, Oliver Schmidt, and Thiemo Voigt. "Cooperative Task Management without Stack Ripping". In: *Proceedings of the 2006 ACM Conference on Embedded Networked Sensor Systems (SenSys '06)*. ACM, 2006, pp. 13–26. DOI: 10.1145/1182807.1182810.

[15] Cormac Flanagan et al. “The essence of compiling with continuations”. In: *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*. PLDI ’93. Albuquerque, New Mexico, USA: Association for Computing Machinery, 1993, pp. 237–247. ISBN: 0897915984. DOI: 10.1145/155090.155113. URL: https://doi.org/10.1145/155090.155113.

[16] Sadjad Fouladi et al. “From laptop to lambda: outsourcing everyday jobs to thousands of transient functional containers”. In: *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC ’19. Renton, WA, USA: USENIX Association, 2019, pp. 475–488. ISBN: 9781939133038.

[17] Saverio Giallorenzo, Fabrizio Montesi, and Marco Peressotti. “Choral: Object-oriented Choreographic Programming”. In: *ACM Trans. Program. Lang. Syst.* 46.1 (Jan. 2024). ISSN: 0164-0925. DOI: 10.1145/3632398. URL: https://doi.org/10.1145/3632398.

[18] Google Cloud Workflows. 2020. URL: https://cloud.google.com/workflows.

[19] Joseph M Hellerstein et al. “Serverless Computing: One Step Forward, Two Steps Back”. In: *ArXiv* abs/1812.03651 (2018). URL: https://api.semanticscholar.org/CorpusID:54462182.

[20] Andrew K. Hirsch and Deepak Garg. “Pirouette: higher-order typed functional choreographies”. In: *Proc. ACM Program. Lang.* 6.POPL (Jan. 2022). DOI: 10.1145/3498684. URL: https://doi.org/10.1145/3498684.

[21] Instagram. *LibCST: A Concrete Syntax Tree Parser and Serializer Library for Python*. https://github.com/Instagram/LibCST. Accessed: 2026-05-20. 2019.

[22] Zhipeng Jia and Emmett Witchel. “Boki: Towards Data Consistency and Fault Tolerance with Shared Logs in Stateful Serverless Computing”. In: *ACM Trans. Comput. Syst.* 42.3–4 (Nov. 2024). ISSN: 0734-2071. DOI: 10.1145/3653072. URL: https://doi.org/10.1145/3653072.

[23] Eric Jonas et al. “Cloud Programming Simplified: A Berkeley View on Serverless Computing”. In: *ArXiv* abs/1902.03383 (2019). URL: https://api.semanticscholar.org/CorpusID:60440467.

[24] Andrew Kennedy. “Compiling with continuations, continued”. In: *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*. ICFP ’07. Freiburg, Germany: Association for Computing Machinery, 2007, pp. 177–190. ISBN: 9781595938152. DOI: 10.1145/1291151.1291179. URL: https://doi.org/10.1145/1291151.1291179.

[25] Maxwell Krohn, Eddie Kohler, and M. Frans Kaashoek. “Events Can Make Sense”. In: *Proceedings of the 2007 USENIX Annual Technical Conference*. USENIX Association, 2007, pp. 1–14. URL: https://www.usenix.org/events/usenix07/tech/full_papers/krohn/krohn.pdf.

[26] E.A. Lee. “The problem with threads”. In: *Computer* 39.5 (2006), pp. 33–42. DOI: 10.1109/MC.2006.180.

[27] David H. Liu et al. “Doing More with Less: Orchestrating Serverless Applications without an Orchestrator”. In: *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA: USENIX Association, Apr. 2023, pp. 1505–1519. ISBN: 978-1-939133-33-5. URL: https://www.usenix.org/conference/nsdi23/presentation/liu-david.

[28] Fabrizio Montesi and Nobuko Yoshida. "Compositional Choreographies". In: *CONCUR 2013 – Concurrency Theory*. Ed. by Pedro R. D'Argenio and Hernán Melgratti. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 425–439. ISBN: 978-3-642-40184-8.

[29] Greg Pettyjohn et al. "Continuations from generalized stack inspection". In: *SIGPLAN Not.* 40.9 (Sept. 2005), pp. 216–227. ISSN: 0362-1340. DOI: 10.1145/1090189.1086393. URL: https://doi.org/10.1145/1090189.1086393.

[30] G.D. Plotkin. "Call-by-name, call-by-value and the λ-calculus". In: *Theoretical Computer Science* 1.2 (1975), pp. 125–159. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(75)90017-1. URL: https://www.sciencedirect.com/science/article/pii/0304397575900171.

[31] François Pottier and Nadji Gauthier. "Polymorphic typed defunctionalization". In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '04. Venice, Italy: Association for Computing Machinery, 2004, pp. 89–98. ISBN: 158113729X. DOI: 10.1145/964001.964009. URL: https://doi.org/10.1145/964001.964009.

[32] Kyriakos Psarakis et al. "Styx: Transactional Stateful Functions on Streaming Dataflows". In: *Proc. ACM Manag. Data* 3.3 (June 2025). DOI: 10.1145/3725363. URL: https://doi.org/10.1145/3725363.

[33] Python Software Foundation. *pickle — Python object serialization*. https://docs.python.org/3/library/pickle.html. Accessed: 2026-06-05.

[34] Sheng Qi, Xuanzhe Liu, and Xin Jin. "Halfmoon: Log-Optimal Fault-Tolerant Stateful Serverless Computing". In: *Proceedings of the 29th Symposium on Operating Systems Principles*. SOSP '23. Koblenz, Germany: Association for Computing Machinery, 2023, pp. 314–330. ISBN: 9798400702297. DOI: 10.1145/3600006.3613154. URL: https://doi.org/10.1145/3600006.3613154.

[35] John C. Reynolds. "Definitional Interpreters for Higher-Order Programming Languages". In: *Proceedings of the ACM Annual Conference - Volume 2*. Association for Computing Machinery, 1972, pp. 717–740. DOI: 10.1145/800194.805852.

[36] Yury Selivanov. *PEP 492 – Coroutines with async and await syntax*. Python Enhancement Proposal. Accessed: 2026-04-23. Apr. 2015. URL: https://peps.python.org/pep-0492/.

[37] Gan Shen, Shun Kashiwa, and Lindsey Kuper. "HasChor: Functional Choreographic Programming for All (Functional Pearl)". In: *Proceedings of the ACM on Programming Languages* 7.ICFP (Aug. 2023), pp. 541–565. ISSN: 2475-1421. DOI: 10.1145/3607849. URL: http://dx.doi.org/10.1145/3607849.

[38] Jeff Smits. *libcst-dfa: A Data-Flow Analysis Framework for Python Built on libCST*. Python package, version 0.0.1. Accessed: 2026-06-14. 2026. URL: https://pypi.org/project/libcst-dfa/.

[39] Jeff Smits, Guido Wachsmuth, and Eelco Visser. "FlowSpec: A declarative specification language for intra-procedural flow-Sensitive data-flow analysis". In: *Journal of Computer Languages* 57 (2020), p. 100924. ISSN: 2590-1184. DOI: https://doi.org/10.1016/j.cola.2019.100924. URL: https://www.sciencedirect.com/science/article/pii/S2590118419300474.

[40] Vikram Sreekanti et al. "Cloudburst: stateful functions-as-a-service". In: *Proc. VLDB Endow.* 13.12 (July 2020), pp. 2438–2452. ISSN: 2150-8097. DOI: 10.14778/3407790.3407836. URL: https://doi.org/10.14778/3407790.3407836.

[41] Guy L. Steele Jr. *RABBIT: A Compiler for SCHEME*. Technical Report AI-TR-474. Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1978.

[42] Guy Lewis Steele Jr. and Gerald Jay Sussman. *Lambda: The Ultimate Imperative*. Tech. rep. AI Memo 353. Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1976.

[43] Don Syme, Tomas Petricek, and Dmitry Lomov. "The F# Asynchronous Programming Model". In: *Practical Aspects of Declarative Languages*. Ed. by Ricardo Rocha and John Launchbury. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 175–189. ISBN: 978-3-642-18378-2.

[44] Temporal. 2019. URL: https://temporal.io/.

[45] The Apache Software Foundation. *Stateful Functions: A Platform-Independent Stateful Serverless Stack*. Apache Flink documentation. Accessed: 2026-05-14. 2021. URL: https://nightlies.apache.org/flink/flink-statefun-docs-stable/.

[46] Transaction Processing Performance Council. *TPC Benchmark C Standard Specification, Revision 5.11*. http://www.tpc.org/tpcc/. Accessed: 2026-05-20. 2010.

[47] Markus Völter and Eberhard Wolff. "Event-Driven Programming for Robust Software-Intensive Systems". In: *Proceedings of the 3rd International Workshop on Software Engineering for Resilient Systems (SERENE 2010)*. Vol. 6968. Lecture Notes in Computer Science. Springer, 2010, pp. 166–178. DOI: 10.1007/978-3-642-24124-6_13.

[48] Xuejun Yang et al. "Finding and understanding bugs in C compilers". In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '11. San Jose, California, USA: Association for Computing Machinery, 2011, pp. 283–294. ISBN: 9781450306638. DOI: 10.1145/1993498.1993532. URL: https://doi.org/10.1145/1993498.1993532.

[49] Haoran Zhang et al. "Fault-tolerant and Transactional Stateful Serverless Workflows". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*. USENIX Association, 2020, pp. 1187–1204. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/zhang-haoran.

[50] Wen Zhang et al. "Kappa: a programming framework for serverless computing". In: *Proceedings of the 11th ACM Symposium on Cloud Computing*. SoCC '20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 328–343. ISBN: 9781450381376. DOI: 10.1145/3419111.3421277. URL: https://doi.org/10.1145/3419111.3421277.

[51] Wouter Zorgdrager et al. "Stateful Entities: Object-oriented Cloud Applications as Distributed Dataflows". In: *CoRR* abs/2112.00710 (2021). arXiv: 2112.00710. URL: https://arxiv.org/abs/2112.00710.