Cloud Contraptions
TRAINING & CONSULTING

To discuss this course and customizations:
Call: +1 434-509-6890 or Email: sales@cloudcontraptions.com

Advanced Zig

Class Duration

21 hours of live training delivered over 3 days.

Student Prerequisites

- Completion of Zig Essentials or equivalent working experience with Zig
- Comfort with the Zig toolchain: building, testing, and running projects with `zig build`
- Working knowledge of pointers, slices, structs, error unions, and basic allocator usage
- General systems programming background is helpful but not required

Target Audience

This course is for developers who already write Zig and want to master the parts of the language that make it distinctive: comptime metaprogramming, explicit allocator design, and the new `std.Io` interface introduced in the Zig 0.16 standard library redesign. It is well suited to teams building libraries, services, games, or embedded systems in Zig who need to write code that is fast, testable, and idiomatic on Zig 0.16 and beyond.

Description

Advanced Zig takes working Zig programmers deep into the language and standard library as they exist in 2026. Zig 0.16, released in April 2026, shipped the largest standard library redesign in the language's history, centered on the new `std.Io` interface: I/O capability is now passed as a parameter - just like an `Allocator` - and async/await returned to the language via `std.Io.Evented`, with stackful coroutines, no function coloring, and high-performance backends built on io_uring on Linux and Grand Central Dispatch on macOS. This course covers that new world in depth: writing Io-generic libraries, structuring concurrent code around idempotent await and cancel with the defer-cancel pattern, and understanding the lock-free, thread-safe allocator design underneath.

Beyond I/O, the course is a tour of advanced Zig craftsmanship: comptime metaprogramming with types as first-class values, designing and composing

allocators, error handling strategies that scale beyond a single file, data-oriented design with `MultiArrayList`, explicit SIMD with `@Vector`, and the testing, fuzzing, and coverage tooling highlighted on the 2026 Zig roadmap. Students leave able to make informed performance and architecture decisions and to write Zig that other Zig programmers recognize as idiomatic.

Learning Outcomes

- Use comptime to write generic, reflective, and zero-cost abstractions, and know where comptime evaluation reaches its limits
- Choose, compose, and implement allocators - arena, fixed-buffer, general-purpose, and custom - and detect leaks in tests
- Write libraries and applications against the Zig 0.16 `std.Io` interface, accepting `Io` as a parameter rather than hard-coding an I/O implementation
- Build concurrent programs with `std.Io.Evented` async/await, tasks, and stackful coroutines, backed by io_uring or Grand Central Dispatch
- Apply the defer-cancel pattern and reason about idempotent await and cancel semantics
- Design error sets, `errdefer` cleanup, and diagnostics patterns that scale across large codebases
- Restructure data with `MultiArrayList` and struct-of-arrays layouts for cache-friendly performance
- Vectorize hot loops explicitly with `@Vector` SIMD types
- Test, fuzz, and measure coverage of Zig code in line with the 2026 roadmap tooling, and profile real workloads
- Select build modes deliberately and explain the safety semantics of each

Training Materials

Comprehensive courseware is distributed online at the start of class. All students receive a downloadable MP4 recording of the training.

Software Requirements

Students will need a free, personal GitHub account to access the courseware. Students will need permission to install Zig 0.16, a C toolchain, and Visual Studio Code (with the Zig language extension) on their computers. A Linux environment is recommended for the io_uring portions of the course; macOS

students will use the Grand Central Dispatch backend. If students are unable to configure a local environment, a cloud-based environment can be provided.

Training Topics

Zig in 2026

- Zig 0.16: the largest stdlib redesign in the language's history
- What changed and why: `std.Io`, async/await reintroduced
- The road to 1.0 and the 2026 roadmap: compiler speedups, fuzzing, coverage
- Tracking the 0.17 development cycle
- Reading and navigating the standard library source

Comptime Metaprogramming Deep Dive

- Types as first-class values
- Generic functions and generic data structures
- `@TypeOf`, `@typeInfo`, and reflection
- `comptime` blocks, parameters, and variables
- Generating code and tables at compile time
- Comptime evaluation limits: branch quotas and what cannot run at comptime
- Comparing comptime to macros, templates, and codegen

Allocator Design

- The `Allocator` interface and the philosophy of explicit allocation
- Arena allocators and lifetime-scoped allocation
- Fixed-buffer allocators for embedded and hot paths
- The general-purpose allocator and its lock-free, thread-safe design
- Writing custom allocators: wrapping, instrumenting, and failing on purpose
- Leak detection in tests with `std.testing.allocator`
- Failure-injection testing of allocation paths

The New std.Io In Depth

- `Io` as a parameter: the Allocator pattern applied to I/O
- `std.Io.Evented` and the async/await model
- Stackful coroutines and fibers: no function coloring
- Backends: io_uring on Linux, Grand Central Dispatch on macOS

- Spawning and managing tasks
- Idempotent await and cancel semantics
- The defer-cancel pattern for structured cleanup
- Writing Io-generic libraries that callers can drive with any implementation
- Blocking, threaded, and evented implementations side by side

Error Handling at Scale

- Error sets: inferred, explicit, and merged
- Designing error taxonomies for libraries vs. applications
- `errdefer` and partial-initialization cleanup
- Diagnostics patterns: returning rich error context without exceptions
- Error return traces and debugging strategies

Data-Oriented Design

- Why struct-of-arrays beats array-of-structs on modern hardware
- `std.MultiArrayList` in practice
- Handles and indices instead of pointers
- Memory layout, alignment, and cache behavior
- Case study: refactoring an entity store to SoA

SIMD with @Vector

- `@Vector` types and vector operations
- `@splat`, `@reduce`, `@shuffle`, and `@select`
- Letting the compiler choose vector width vs. fixing it
- Vectorizing a real hot loop and measuring the gain
- When the auto-vectorizer already has you covered

Testing, Fuzzing, and Coverage

- Structuring test blocks across a large codebase
- Built-in fuzzing on the 2026 roadmap: `zig build test --fuzz`
- Coverage tooling and interpreting results
- Testing Io-generic code with test implementations
- Property-style testing patterns in Zig

Performance, Build Modes, and Safety

- Build modes: Debug, ReleaseSafe, ReleaseFast, ReleaseSmall
- Safety semantics: which checks exist in which mode

To discuss this course and customizations:
Call: +1 434-509-6890 or Email: sales@cloudcontraptions.com

- Illegal behavior vs. undefined behavior
- Profiling Zig programs: perf, Tracy, and benchmark harnesses
- Reading optimized output and verifying assumptions

